3.9.4. eval() & repr()

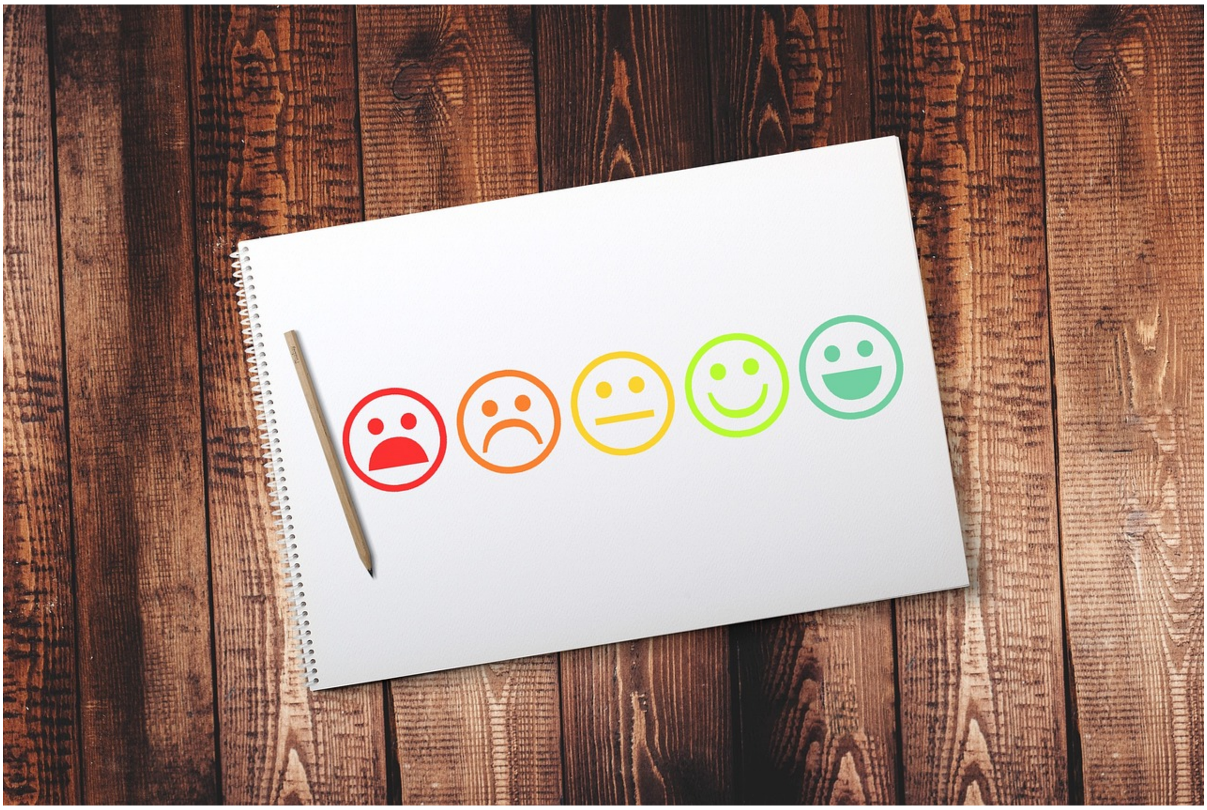

Fig. 3.9.4.1 Image by Tumisu from Pixabay

Note

Outline

1. eval()
    a. Ex1: Calculation
    b. Ex2: Calculation with Variables
    c. Ex3: input()
    d. Ex4: compile()
    e. Ex5: exec()
    f. Ex6a: globals
    g. Ex6b: globals 2
    h. Ex6c: globals 3: Empty
    i. Ex7a: locals
    j. Ex7b: locals 2
    k. Ex8: subprocess()
    l. Ex9: literal_eval()
2. repr()
    a. Ex1: Custom

Note

Roadmap

1. This topic: TryExcept

> myBlock
>
> | Exception Handling | | | | |
> |---|---|---|---|---|
> | try Statement | except Statement | finally Statement | Error Type | with Statement |

2. Course: Python 1
3. Subject: Programming
4. Field
    a. Software Engineering (SE)
    b. Computer Science and Information Engineering (CSIE)
    c. Electrical/Electronics Engineering (EE)

3.9.4.1. eval()

1. The eval() function in Python is used to evaluate a dynamically created Python expression, which can be a string containing Python code.
2. However, be cautious when using eval(), as it can execute arbitrary code and potentially introduce security risks if used with untrusted input.
3. Syntax

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

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

> a. Parse expression
>
> b. Compile it to bytecode
>
> c. Evaluate it as a Python expression
>
> d. Return the result of the evaluation

Important

Security

1. eval() is considered insecure because it allows you (or your users) to dynamically execute arbitrary Python code.
2. For this reason, good programming practices generally recommend against using eval().

3.9.4.1.1. Ex1: Calculation

1. Code+Output

Code **Output**

a. We evaluate a simple mathematical expression "2 + 3 * 4" using eval() and print the result.

Listing 3.9.4.1.1.1 /src/Eval+Repr/Eval/__init__.py

```
'''
author: cph
since:  20130102
'''
# Evaluate a simple mathematical expression
sExpr = "2 + 3 * 4"
iResult = eval(sExpr)
print("Result of expression:", iResult)
```

3.9.4.1.2. Ex2: Calculation with Variables

1. Code+Output

Code **Output**

a. We use eval() with variables by providing a dictionary as the second argument.

b. This allows us to evaluate an expression like “x + y” with values for x and y.

Listing 3.9.4.1.2.1 /src/Eval+Repr/Eval2/__init__.py

```
'''
author: cph
since:  20190102
'''
# Using eval() with variables
x = 5
y = 10
sExpr = 'x + y'
iResult = eval(sExpr, {'x': x, 'y': y})
print('Result of expression:', iResult)
```

3.9.4.1.3. Ex3: input()

1. Code+Output

Code **Output**

a. We demonstrate how eval() can be used to get user input as a Python expression and evaluate it.

b. We wrap the eval() call in a try/except block to handle potential errors, as eval() can raise exceptions for invalid or unsafe input.

Listing 3.9.4.1.3.1 /src/Eval+Repr/Eval3/__init__.py

```
'''
author: cph
since:  20130102
'''
# Using eval() to get user input and execute it
sIn = input("Enter a Python expression: ")
try:
    oResult = eval(sIn)
    print("Result of expression:", oResult)
except Exception as e:
    print("Error:", e)
```

3.9.4.1.4. Ex4: compile()

1. Compile the source into a code or AST object. [Web]

> **Note**
>
> AST
>
> a. AST = Abstract Syntax Tree [抽象語法樹] = Syntax Tree [語法樹]

2. Code objects can be executed by exec() or eval().

3. source can either be a normal string, a byte string, or an AST object.

4. Refer to the ast module documentation for information on how to work with AST objects.

5. Syntax

```
compile(source, filename, mode, flags=0, dont_inherit=False, optimize=- 1)
```

6. Code+Output

Code Output

Listing 3.9.4.1.4.1 /src/Eval+Repr/Eval4/__init__.py

```
'''
author: cph
since:  20130102
'''
import math

sCode = compile('3 + 4', '<string>', 'eval')
print(eval(sCode))

sCode = compile('3 * 3 * math.pi', '<string>', 'eval')
print(eval(sCode))
```

3.9.4.1.5. Ex5: exec()

1. This function supports dynamic execution of Python code. [Web]

2. object must be either a string or a code object.

3. If it is a string, the string is parsed as a suite of Python statements which is then executed.

4. Syntax

```
exec(object, globals=None, locals=None, /, *, closure=None)
```

5. Code+Output

Code Output

Listing 3.9.4.1.5.1 /src/Eval+Repr/Eval5/__init__.py

```
'''
author: cph
since:  20190102
'''
sCode = '''
def greet(sIn):
    return f'Hello, {sIn}...'

print(greet('Python'))
'''

exec(sCode)
eval(sCode)
```

3.9.4.1.6. Ex6a: globals

1. Code+Output

Code Output

Listing 3.9.4.1.6.1 /src/Eval+Repr/EvalGlobals/__init__.py

```
'''
author: cph
since:  20190102
'''
x = 3  # Global variable
y = 4  # Global variable
print(eval('x + 5', {'x': x}))
print(eval('x + y', {'x': x, 'y': y}))
print(eval('x + y', {'x': x}))
```

3.9.4.1.7. Ex6b: globals 2

1. Code+Output

Code Output

Listing 3.9.4.1.7.1 /src/Eval+Repr/EvalGlobals2/__init__.py

```
'''
author: cph
since:  20190102
'''
x = 3  # Global variable
y = 4  # Global variable
print(eval('x + y + z', {'x': x, 'y': y, 'z': 5}))
print(f'z: {z}')
```

3.9.4.1.8. Ex6c: globals 3: Empty

1. Code+Output

Code Output

a. We set globals argument to {}, that is, we send an empty dictionary as globals.

Listing 3.9.4.1.8.1 /src/Eval+Repr/EvalGlobals3/__init__.py

```
'''
author: cph
since:  20190102
'''
print(eval('max((4, 3, 2, 1))'))
print(eval('max((4, 3, 2, 1))', {}))
print(eval('max((4, 3, 2, x))', {'x': 5}))
print(eval('min([4, 3, 2, 1])', {}))
```

```
print(eval('sum({4, 3, 2, 1})', {}))
print(eval('sorted([4, 3, 2, 1])', {}))
```

3.9.4.1.9. Ex7a: locals

1. Code+Output

Code Output

Listing 3.9.4.1.9.1 /src/Eval+Repr/EvalLocals/__init__.py

```
'''
author: cph
since:  20190102
'''
print(eval('x + 5', {'x': 3}))
print(eval('x + y', {'x': 3, 'y': 4}))
print(eval('x + y', {'x': 3}, {'y': 5}))
print(eval('x + y', {'x': 3}))
```

3.9.4.1.10. Ex7b: locals 2

1. Code+Output

Code Output

Listing 3.9.4.1.10.1 /src/Eval+Repr/EvalLocals2/__init__.py

```
'''
author: cph
since:  20190102
'''
x = 3
print(eval('x + 5', {'x': x}))
print(eval('x + y', {'x': x}, {'y': 4}))
print(eval('x + y', {'x': 13}, {'y': 4}))
print(eval('x + y', {'x': x}, {'y': 4}))
print(eval('x + y', {'x': x}, {'y': 4, 'x': 6}))
print(eval('x + y', {'x': x}, locals={'y': 6}))
```

3.9.4.1.11. Ex8: Complex Expression

1. Code+Output

Code Output

Listing 3.9.4.1.11.1 /src/Eval+Repr/EvalComplexExpr/__init__.py

```
'''
author: cph
since:  20190102
'''
import subprocess
```

```
# Echo a string
print(eval("subprocess.getoutput('echo Hello, Python...')"))

# Open Chrome browser in Windows
sPath = 'C:\Program Files (x86)\Google\Chrome\Application\chrome.exe'
sPara = '--kiosk'
print(eval("subprocess.call([sPath, sPara])"))
```

3.9.4.1.12. Ex9: literal_eval()

1. Code+Output

Code **Output**

Listing 3.9.4.1.12.1 /src/Eval+Repr/EvalLiteral/__init__.py

```
'''
author: cph
since:  20190102
'''
from ast import literal_eval

print(literal_eval('1.2'))
print(literal_eval('(1, 2, 3, 4)'))
print(literal_eval('[1, 2, 3, 4]'))
print(literal_eval("{1: 'a', 2: 'b', 3: 'c'}"))
print(literal_eval('sum([1, 2]) + 3 - 4'))
```

Important

Summary: eval()

1. Remember that using eval() with untrusted or unsanitized input can be dangerous, as it can execute arbitrary code.
2. Be cautious and avoid using it with input from untrusted sources.

See also

Reference

1. Leodanis Pozo Ramos, Python eval(): Evaluate Expressions Dynamically, RealPython, 20230801. [Web]

3.9.4.2. repr()

1. The repr() function in Python is used to generate a string representation of an object, which, if passed to the eval() function, should return an object with the same value.
2. It's primarily used for debugging and inspection purposes.

3.9.4.2.1. Ex1: Custom

1. Code+Output

Code Output

a. We define a custom class MyClass with a __repr__() method.

b. This method returns a string representation of the object, including its attributes.

c. We create an instance of MyClass called obj with specific values for x and y.

d. We use repr(obj) to get a string representation of the obj object, which is provided by the __repr__() method.

e. We print the string representation, and we can see that it matches the format specified in the __repr__() method.

f. We use eval() to recreate the object recreated_obj from its string representation.

g. This demonstrates that the repr() string can be used to reconstruct an object.

Listing 3.9.4.2.1.1 /src/Eval+Repr/Repr/__init__.py

```python
'''
author: cph
since:  20180102
'''
class MyClass:  # Define a custom class
    def __init__(self, x, y):
        self.x = x
        self.y = y

    # Implement a custom __repr__() method
    def __repr__(self):
        return f"MyClass(x={self.x}, y={self.y})"

obj = MyClass(5, 10)    # Create an instance of the custom class

# Use repr() to get a string representation of the object
oRepr = repr(obj)
print(oRepr)

# Use eval() to recreate the object from its repr() string
oEval = eval(oRepr)
print(oEval)
```

Important

Summary: repr()

1. Using repr() with custom classes allows you to provide meaningful representations of your objects, making it easier to understand their state and content during debugging or when displaying information about them.

Note

1. Start: 20170719
2. System Environment:

Listing 3.9.4.2.1.2 requirements.txt

```
sphinx==7.1.2                           # Sphinx
graphviz>=0.20.1                        # Graphviz
sphinxbootstrap4theme>=0.6.0            # Theme: Bootstrap
sphinx-material>=0.0.35                 # Theme: Material
sphinxcontrib-plantuml>=0.25            # PlantUML
sphinxcontrib.bibtex>=2.5.0             # Bibliography
sphinx-autorun>=1.1.1                   # ExecCode: pycon
sphinx-execute-code-python3>=0.3        # ExecCode
btd.sphinx.inheritance-diagram>=2.3.1   # Diagram
sphinx-copybutton>=0.5.1                # Copy button
sphinx_code_tabs>=0.5.3                 # Tabs
sphinx-immaterial>=0.11.3               # Tabs

#------------------------------------------------------
#-- Library Upgrade Error by Library Itself
#   >> It needs to fix by library owner
#   >> After fixed, we need to try it later
#------------------------------------------------------
pydantic==1.10.10                       # 2.0: sphinx compiler error, 20230701

#------------------------------------------------------
#-- Minor Extension
#------------------------------------------------------
sphinxcontrib.httpdomain>=1.8.1         # HTTP API

#sphinxcontrib-blockdiag>=3.0.0         # Diagram: block
#sphinxcontrib-actdiag>=3.0.0           # Diagram: activity
#sphinxcontrib-nwdiag>=2.0.0            # Diagram: network
#sphinxcontrib-seqdiag>=3.0.0           # Diagram: sequence

#------------------------------------------------------
#-- Still Wait For Upgrading Version
#------------------------------------------------------

#------------------------------------------------------
#-- Still Under Testing
#------------------------------------------------------
#numpy>=1.24.2                          # Figure: numpy

#------------------------------------------------------
#-- NOT Workable
#------------------------------------------------------
#sphinxcontrib.jsdemo==0.1.4    # ExecCode: Need replace add_js_file()
#jupyter-sphinx==0.4.0          # ExecCode: Need gcc compiler
#sphinxcontrib.slide==1.0.0     # Slide: Slideshare
#hieroglyph==2.1.0              # Slide: make slides
#matplotlib>=3.7.1              # Plot: Need Python >= v3.8
#manim==0.17.2                  # Diagram: scipy, numpy need gcc
#sphinx_diagrams==0.4.0         # Diagram: Need GKE access
#sphinx-tabs>=3.4.1                     # Tabs: Conflict w/ sphinx-material
```