Documentation of Automated_Error_Checks.py

by CyberCoral

June 11, 2025

Contents

1 Introduction to the program.

The program, written in Python3 and using the regex module **re**, is a python error handler for general use. It means this program will let you cover more easily each condition and type check you would need.

Instead of using various if-else statements, which occupy many lines of code, you can use that module and its functions to compact that part of any secure code.

Listing 1: Credits for the author.

```
# ver. Wed/11/Jun/2025
#
# Made by: CyberCoral
# ----------------------------------------------------------
# Github:
# https://www.github.com/CyberCoral
#
```

There are three main sections that this documentation will cover:

1.1 Simple functions.

In this section, the basics of the program will be covered. It will view two main programs:

ErrorTypeFinder() and ConditionCheck(). 2.1

It also will show a quality of life converter called BinaryToTruthList(). 2.3

2

1.2 Complex / Automated functions.

In this part, there are the advanced versions of the three basic functions, expanded to cover more than 1 variable at a time.

3

1.3 Use in Python3 example.

Last but not least, there will be an example of how you can use the module and its functions to save code.

4

2 Simple functions: BinaryToTruthList(), ErrorTypeFinder(), ConditionCheck()

The core functions of the module are ErrorTypeFinder() and ConditionCheck():

2.1 ErrorTypeFinder()

Listing 2: The code for the function (lines 35 to 109 of the program).

```
###
### This program will be used to check common variable types
###

def ErrorTypeFinder(var, conditions):
    '''
    This program checks for variable's type
    according to conditions' truth values.
    The order of validation is this one:
    int -> float -> complex -> tuple
    -> list -> dict.          (str)

    Any other more input will not be considered.
    If there are less than 6 inputs, all the other
    ones are False by default, except with str.
    In that case, you have the max of 7 inputs.
    '''

    dict_condition = {}
    condition = ["int","float","complex","tuple","list","dict","str"]

    if isinstance(conditions, list) != True:
        return ErrorTypeFinder(conditions,BinaryToTruthList("000010"))

    for i in range(len(conditions)):
        if isinstance(conditions[i], bool) != True:
            raise SyntaxError("The conditions' value ({}) type ({}) is invalid."

    if len(conditions) <= len(condition):
        for i in range(len(condition) - len(conditions)):
            conditions.append(False)
    elif len(conditions) >= len(condition):
        conditions = conditions[0:len(condition)]

    if conditions.count(True) > 1:
        raise SyntaxError("A variable cannot be two types at the same time.")
```

```
    elif conditions.count(False) == len(conditions):
        raise SyntaxError("There has to be a condition that is met.")

    elif len(conditions) >= 7:
        if isinstance(var, str) != True and conditions.index(True) == 6:
            raise TypeError("{} is supposed to be {}".format(var, str))
        elif isinstance(var, str) != True and conditions.index(True) != 6:
            pass

        elif isinstance(var, str) == True and conditions.index(True) == 6:
            return True
        else:
            raise TypeError("{} is supposed to be {}".format(var, str))

    elif isinstance(var, str) == True and len(conditions) < 7:
        raise TypeError("{} is supposed to be {}, but cannot be valued because o
    else:
        del conditions[6:]

    if len(conditions) <= len(condition):
        for i in range(len(condition) - len(conditions)):
            conditions.append(False)
    elif len(conditions) >= len(condition):
        conditions = conditions[0:len(condition)]

    for j in range(len(condition)):
        dict_condition.update({condition[j]: conditions[j]})

    conditions = []

    for k in range(len(condition)):

        conditions.append(f"""if (lambda var, k, condition, dict_condition: Fals
raise TypeError('''Type of {var} is supposed to be {condition[list(dict_conditio
        exec(compile(conditions[0],"<string>","exec"))
        del conditions[0]

    return True
```

First, this function has two arguments:

1. var (the variable you introduce)
2. condition (a list of True/False values that determines what type var should be)

Then, let's check the order of types in the function:

```
int -> float -> complex -> tuple
-> list -> dict.          (str)
```

The types follow this order:

1. int
2. float
3. complex
4. tuple
5. list
6. dict
7. (str) (opcional)

There can be at most 7 boolean values that determine types by default in the program (although you can add more if you reprogram it).

If there are more than 7 values (or the number of elements in condition), those will be omitted. You can only choose one type to check at a time without getting an error.

After a lot of checks for the conditions variable, a general exec() check will be executed, which contains the next logic:

```
    for k in range(len(condition)):

        conditions.append(f"""if (lambda var, k, condition, dict_condition: Fals
raise TypeError('''Type of {var} is supposed to be {condition[list(dict_conditio
        exec(compile(conditions[0],"<string>","exec"))
        del conditions[0]
```

Here, although in a difficult way, the program checks for each element if the variables is or not its adequate type.

If not, the program returns a TypeError. In this context, exec() is not a security issue, because the string which is passed through it is regulated and has been checked.

Next, for more general use cases, ConditionCheck():

2.2 ConditionCheck()

```
###
### This program checks for conditions on the variable var with 2 different sever
###
def ConditionCheck(var, conditions: str, severity_mode: int = 1):
    '''
    The function checks for str conditions
    and if they are correct or not.
    Also, depending on severity_mode, it can
    return different types of errors:

    - severity_mode = 1: False if the conditions
    are not met.

    - severity_mode = 2: SyntaxError if the
    conditions are not met.

    The structure of conditions is the next one:
    "<condition> && <condition>" for "and" structures.
    "<condition> || <condition>" for "or" structures.

    All the conditions must have "var" in them.
    '''

    if isinstance(conditions, str) != True:
        raise TypeError("conditions must be a str.")
    elif re.search(".*var",conditions) == None:
        raise SyntaxError("conditions must contain var at least once.")

    if isinstance(severity_mode, int) != True:
        raise TypeError("severity_mode must be an int.")

    c = "".join([str(i) for i in conditions]).replace("&&"," and ").replace("||"
    l = []

    # Fixed an issue with variables like 'var' or 'l' not being defined
    # inside exec(), now all the variables become global inside exec(),
    # including 'var' and 'l'.
    globalvars = {k:v for k,v in globals().items()}
    globalvars.update({"var":var, "l": l})


    match severity_mode:
```

```
case 1:

    try:
        exec(compile(f"b = {c}\nl.append(b)","<string>","exec"),globalva
        b = l[0]
        if b == False:
            return False
        return True
    except SyntaxError:
        return False

case 2:

    try:
        exec(compile(f"b = {c}\nl.append(b)","<string>","exec"),globalva
        b = l[0]
        if b == False:
            raise ValueError(f"The conditions ({c}) are not met with var
        return True
    except SyntaxError:
        raise SyntaxError(f"The conditions ({c}) do not make sense, they

case _:

    raise OSError(f"This severity mode ({severity_mode}) is not included
```

Instead of using two arguments, ConditionCheck() utilizes 3, which are:

1. var (the variable)
2. conditions (a string with all the conditions var has to follow)
3. severity_mode (it indicates the program what it has to do when an error occurs)

There are three main features the function presents:

- The conditions must not syntax errors (they work exactly like other error handling).
- You can fuse one or more conditions with && (AND operator) or || (OR operator).
- You can adjust the severity mode to 1 (returns False if condition is not met) or 2 (raises TypeError when condition is not met).

The last function is this one:

2.3 BinaryToTruthList()

```
###
### This program makes a truth list out of a binary string of characters (made o
###

def BinaryToTruthList(binary):
    '''
    This program returns a list consisted of
    Truth and False based on a string of
    ONLY 0s and 1s (if this rule is broken, the
    program will return a SyntaxError).
    '''
    binary = str(binary)
    try:
        binary = [int(i) for i in binary]
    except ValueError:
        raise SyntaxError("{} is an invalid string from which to create a truth
    truth_list = []
    for j in range(len(binary)):
        if binary[j] not in [0,1]:
            raise SyntaxError("{} is an invalid character of a binary string.".f
        else:
            truth_list.append(bool(binary[j]))
    return truth_list
```

It transforms a binary number (made with 1s and 0s) to a list with True or False values.

3 Complex functions: AutomatedBinaryToTruthList(), AutomatedErrorTypeFinder(), AutomatedConditionCheck()

These functions are automated versions of the previous basic ones, so to understand these one learning the simple ones is a must.

3.1 AutomatedBinaryToTruthList()

```
###
### This program makes an
### automated conversion of truth lists out of a binary string of characters (ma
###

def AutomatedBinaryToTruthList(binaries: list):
    '''
    This program returns a list of lists consisted of
    Truth and False based on a string of
    ONLY 0s and 1s (if this rule is broken, the
    program will return a SyntaxError).
    '''
    if isinstance(binaries, list) != True:
        raise TypeError("binaries should be a list")

    a = []
    for i in range(len(binaries)):
        a.append(BinaryToTruthList(binaries[i]))

    return a
```

In this function, the argument is a list with multiple binary numbers, which are converted into boolean value lists.

3.2 AutomatedErrorTypeFinder()

```
###
### This program makes an
### automated check for common variable types on variables.
###

def AutomatedErrorTypeFinder(variables: list, condition_batch: list):
    '''
    This program automatically checks for
    types of variables with condition_batch.
    It follows the same rules of ErrorTypeFinder().
    '''
    if isinstance(variables, list) != True:
        raise TypeError("variables must be a list")
    elif isinstance(condition_batch, list) != True:
        raise TypeError("condition_batch must be a list")

    if len(variables) != len(condition_batch):
        raise SyntaxError("There must be the same number of elements for variabl

    for i in range(len(variables)):
        condition = condition_batch[i]
        if isinstance(condition, list) != True:
            try:
                condition = BinaryToTruthList(condition)
            except SyntaxError:
                raise TypeError("condition_batch[{}] ({}) is not a truth list or

        ErrorTypeFinder(variables[i], condition)

    return True
```

The arguments in this function are lists, which are these ones:

1. variables (the list of variables)

2. condition_batch (a list with multiple boolean value lists OR binary numbers)

For each variable, there must be a condition list. If that is not the case, it is a SyntaxError.

Then, each variable will be checked and it will return True if no error occurs.

3.3 AutomatedConditionCheck()

```
###
### This program makes an
### automated check for conditions on variables with 2 different types of severit
###
def AutomatedConditionCheck(variables: list , condition_batch: list , severity_mod
    '''
    This program automatically checks for
    conditions in variables with
    condition_batch and with
    severity_modes.
    It follows the same rules of
    ConditionCheck().
    '''
    if isinstance(variables, list) != True:
        raise TypeError("variables must be a list")
    elif isinstance(condition_batch, list) != True:
        raise TypeError("condition_batch must be a list")
    elif isinstance(severity_modes, list) != True:
        raise TypeError("severity_modes must be a list")

    if len(variables) != len(condition_batch) or len(variables) != len(severity_m
        raise SyntaxError("There must be the same number of elements for variabl

    for i in range(len(condition_batch)):
        if isinstance(condition_batch[i], str) != True:
            raise TypeError("conditions must be a str.")
        elif re.search(".*var",condition_batch[i]) == None:
            raise SyntaxError("conditions must contain var at least once.")

    for i in range(len(severity_modes)):
        if isinstance(severity_modes[i], int) != True:
            raise TypeError("severity_mode must be an int.")

    results = []

    for i in range(len(variables)):
        results.append(ConditionCheck(variables[i], condition_batch[i], severity_

    return results
```

There are three arguments in this function:

1. variables (a list with all the variables)
2. condition_batch (a list with all the conditions)

There must be a condition and severity mode per variable, if that is not the case, it is a SyntaxError.

The results will be returned if there are no errors.

4 Use in Python3 code example.

```
import Automated_Error_Checks as AEC # Shorten the name.

def natural_inv(number):
  AEC.AutomatedErrorTypeFinder([number],[1]) # number must be an int.
  AEC.AutomatedConditionCheck([number],["var != 0"],[2]) # For edge case 0.
  return 1 / number

a = natural_inv(2)
b = natural_inv(0) # return ValueError
```

ValueError: The conditions (var != 0) are not met with var = 0.

It successfully predicts the edge case, without making another error take its place.

You only need two lines of code for normal error handling.

5 Credits

The author's Github page: https://github.com/CyberCoral
The repository url: https://github.com/CyberCoral/Error_Type_Checker