**Built-in functions**

jueves, 6 de octubre de 2022

1:30 p. m.

Documentation: <https://docs.python.org/3/library/functions.html>

**round()**: rounds up a float to the nearest integer, and can take up to 2 arguments, the first would be the floating number to round, and the second would be the number of decimals are needed on the result.

**set()**: Takes an iterable as parameter an iterable (sequence or collection) and return a set of unique values in it.

**filter()**: This method works similar to map(), it takes two arguments: 1. a function, 2. an iterable. And the function runs the function to each element of the iterable and if returned true it appends it to an iterator object and if returned false it would leave that element out of the iterable.

The result of this function could be easily transformed into something printable like a list, string, tuple, dictionary, etc…

**any()**: This function takes an iterable as parameter and will return True if any element of the iterable is True. If the iterable is empty it will return False.

**all()**: This function takes an iterable as parameter and will return True if all elements of the iterable are True. If the iterable is empty it will return True.

**eval()**: This function takes up to 3 argument, which only the first one is necessary, the other two are optional. Up to this point of my learning the used I could draw from this function is that it will evaluate literally the string given as if it were actual code.

The example for this code will be the one on I got to know it. The task was to check all the is\_something string methods in a given string. Someone, in hackerrank discussions proposed a solution based on this function.

s = 'qA2'

for fct in ['alnum', 'alpha', 'digit', 'lower', 'upper']:

        print(any([ eval(f'char.is{fct}()') for char in s ]))

The 'eval(f'char.is{fct}()')' part, will check for each element of the list ['alnum', 'alpha', 'digit', 'lower', 'upper'] check and actually evaluate the literal to see whether the character belongs to any of the validators

**repr()**: This function returns a string printable representation of an object. It's pretty similar to the **str**() function, but they differ in purpose:

\_\_ **str**()\_\_ is meant to be readable

\_\_ **repr**()\_\_ is meant to be unambiguous

Here an example:

import datetime

a = datetime.datetime.utcnow()

b = str(a)

print ('str(a): {}'.format(str(a)))

print ('str(b): {}'.format(str(b)))

print ('repr(a): {}'.format(repr(a)))

print ('repr(b): {}'.format(repr(b)))

The output would be:

str(a): 2022-10-13 00:31:03.731001

str(b): 2022-10-13 00:31:03.731001

repr(a): datetime.datetime(2022, 10, 13, 0, 31, 3, 731001)

repr(b): '2022-10-13 00:31:03.731001'

Whenever working with **eval()** function is better to work with **repr()** than with **str**():

**str**():

print( eval(str(a)) )

The output would be:

Se una excepci6n: x 
invalid syntax line I) 
File Problems. py", 
print( ) 
line 435, in 

But with **repr():**

print( eval(repr(a)) )

The output would be:

2022-10-13 00:40:37.035585

**enumerate*(*** *iterable, start=0* **)**: This function returns an enumerate type object, that contains tuples each one with the count as number of the item, and the item that comes from the iterable passed in the argument. The second argument is optional and overwrites he first number of the count, so it could start counting the items from 0 (default, non-passed argument), 10 or 100.

To better see what enumerate( ) will return:

a = ['X', 'Y', 'Z']

b = enumerate(a)

print(list(b)) = [(0, 'X'), (1, 'Y'), (2, 'Z')]

*The list( ) cast is needed since an enumerate instance is not printable.*

Is recommended to use it for iterating through and sequence to not do the count out of the for loop thing as follows:

a = [1, 4, 8]

count = 0

for i in a:

    print (count, i)

    count += 1

Output:

0 1

1 4

2 8

With enumerate( ) is easier:

for count, i in enumerate(a):

    print(count, i)

Output:

0 1

1 4

2 8

**sorted*(*** *iterable, /, \*, key=None, reverse=False* **) -** [Library's How To](https://docs.python.org/3/howto/sorting.html#sortinghowto): The first thing to note is that this function differs from the *list.*sort( ) built-in method in the fact that the list method alter the actual list object, but this function build a new list with the iterable and criteria from the arguments.

This function returns a list.

The key is used to compare and the reverse to shift from ascendant to descending order.

#Shifting the order of a list with sorted()

l = ['A', 'B', 'C', 'D']

l4 = sorted(l,key=None, reverse=True)

print(l4) = ['D', 'C', 'B', 'A']

Different from *list.*sort( ), the function works with all kinds of iterables.

Key functions

The *key* argument works as a function that is to be apply to each item on the iterable to compare to one another. **lambda** usually works a lot for this

student\_tuples = [

    ('john', 'A', 15),

    ('jane', 'B', 12),

    ('dave', 'B', 10),

]

print(sorted(student\_tuples, key= lambda student: student[2])) = [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]

# or with descending order

student\_tuples = [

    ('john', 'A', 15),

    ('jane', 'B', 12),

    ('dave', 'B', 10),

]

print(sorted(student\_tuples, key= lambda student: student[2], reverse=True))

**min*(*** *iterable, /, \*, default*, *key=None* **):** This function could either receive an iterable as argument as well as positional arguments to be compared and return only the lowest value within. The *key* argument works the same as for the sorted() function. And finally, if the iterable passed is empty a ValueError will be raised. To avoid this error the *default* optional argument has the job to be the return value is the iterable is a void one. THIS WORKS THE SAME FOR **max*(* )** function.

**zip*(*** *iterables, strict=False* **):** This function essentially take iterables and in order from left to right pair them into tuples and return an iterator (a 'zip' class object) made of those tuples.

x = ('A', 'B', 'C')

y = ('D', 'E', 'F')

z = ('G', 'H', 'I')

print(type(zip(x,y,z))) = <class 'zip'>

print(list(zip(x,y,z))) = [('A', 'D', 'G'), ('B', 'E', 'H'), ('C', 'F', 'I')]

The function may have no arguments and will return an empty iterator, or if takes only one iterable it will deliver the iterator filled with only one element tuple. The iterables don't necessarily need to be the same length but the function will stop when the shorter gets exhausted.

x = ('A', 'B', 'C')

y = ('D', 'E')

print(list(zip(x,y))) = [('A', 'D'), ('B', 'E')]

If the *'strict'* argument is set to 'True' The function will return a ValueError pointing that the two arguments must be same size.

A zip class object can be unpacked using the \* operator:

zipped = zip(x,z)

x2, z2 = zip(\*zipped)

print(x2, z2) = ('A', 'B', 'C') ('G', 'H', 'I')

Matrix transposition with zip( )

This a practical example on how this function works. If one would like to turn

matrix = [

    [1, 2, 3, 4],

    [5, 6, 7, 8],

    [9, 10, 11, 12],

]

Into

trans = [

[1, 5, 9]

[2, 6, 10]

[3, 7, 11]

[4, 8, 12]

]

There could be many ways, but two are listcomp and nested for loops:

Listcomp

trans = [ [row[i] for row in matrix] for i in range(4) ]

Nested fors

trans = []

for i in range(4):

    temp = []

    for row in matrix:

        temp.append(row[i])

    trans.append(temp)

But the same result is achievable through **zip**( ) and the \* operator:

trans\_temp = list(zip(\*matrix))

itertools.**zip\_longest**( *\*iterables, fillvalue=None* )

Source: <https://docs.python.org/3/library/itertools.html#itertools.zip_longest>

While the **zip**( ) function requires that all the iterables have the same length, this method from the itertools module would take the longest iterable to exhausted pairing it with the rest of the iterables and filling with None by default if no filling value is specified

from itertools import zip\_longest

iter1 = ['A', 'B', 'C', 'D', 'E']

iter2 = ['x', 'y', 'z']

iter3 = [1, 2, 3]

iter4 = list(zip\_longest(iter1, iter2, iter3, fillvalue=None))

print(iter4) = [

('A', 'x', 1),

('B', 'y', 2),

('C', 'z', 3),

('D', None, None),

('E', None, None)

]

**breakpoint( ):** This function is quite specific for the debugging context, it create breakpoint throughout the code's execution in order to make adjustments on the fly. Here's a [tutorial](https://www.youtube.com/watch?v=aZJnGOwzHtU) of this built-in function.

**callable( ):** This function checks whether an object can be call like a function, it will return True or False if is callable. This checks if the object has the \_\_call\_\_() method implemented. This will return True to built-in functions, classes and methods, and False to

class Car:

    def \_\_init\_\_(self):

        pass

sail = Car()

print(callable(sail)) #False

A simple instance of a Class is not a callable, but, if it would have the \_\_call\_\_() method implemented it would have return True

class Car:

    def \_\_init\_\_(self):

        pass

    def \_\_call\_\_(self):

pass

sail = Car()

print(callable(sail))   #True

**Open( ):**

**Source:** <https://docs.python.org/3/tutorial/inputoutput.html#tut-files>

**This function checks whether an object can be call like a function, it will return True or False if is callable. This checks if the object has the \_\_call\_\_() method implemented. This will return True to built-in functions, classes and methods, and False to**