**Basic list comprehensions**

For this task, you will have to create a bag-of-words representation of the spam email stored in the spam variable (you can explore the content using the shell). Recall that bag-of-words is simply a counter of unique words in a given text. This representation can be further used for text classification, *e.g.* for spam detection (given enough training examples).

We created a small auxiliary function create\_word\_list() to help you split a string into words, *e.g.* applying it to 'To infinity... and beyond!' will return ['To', 'infinity', 'and', 'beyond'].

**Instructions 1/4**

**25 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Convert the text to lower case and create a word list.

# Convert the text to lower case and create a word list

words = create\_word\_list(spam.lower())

**Instructions 2/4**

**25 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Create a set that will store only unique words from the list.

# Convert the text to lower case and create a word list

words = create\_word\_list(spam.lower())

# Create a set storing only unique words

word\_set = set(words)

**Instructions 3/4**

**25 XP**

* [3](javascript:void(0))
* [4](javascript:void(0))
* Using list comprehension, create a dictionary that counts a word appearance in the word list.

# Convert the text to lower case and create a word list

words = create\_word\_list(spam.lower())

# Create a set storing only unique words

word\_set = set(words)

# Create a dictionary that counts each word in the list

tuples = [(word, words.count(word)) for word in word\_set]

word\_counter = dict(tuples)

**Instructions 4/4**

**25 XP**

* [4](javascript:void(0))
* Print words that appear in the word\_counter more than once.

# Convert the text to lower case and create a word list

words = create\_word\_list(spam.lower())

# Create a set storing only unique words

word\_set = set(words)

# Create a dictionary that counts each word in the list

tuples = [(word, words.count(word)) for word in word\_set]

word\_counter = dict(tuples)

# Printing words that appear more than once

for (key, value) in word\_counter.items():

if value > 1:

print("{}: {}".format(key, value))

<script.py> output:

you: 3

to: 4

team: 2

the: 4

verify: 2

account: 3

administration: 2

service: 2

your: 4

our: 2

Great! You can observe that one of the most frequent words in the email are account, service, administration, and verify. This might be a security email making sure an unusual login attempt was made by the owner of the account... Or it can be a phishing email!

**Prime number sequence**

A prime number is a positive number that is divisible by 1 or itself (*e.g.* 3, 7, 11 *etc.*). However, 1 is not a prime number.

Your task is, given a list of candidate numbers cands, to filter only prime numbers in a new list primes.

But first, you need to create a function is\_prime() that returns True if the input number nn is prime or False, otherwise. A number is prime if it is not divisible by any integer number from 2 to n−−√n (any number nn is not divisible by anything higher than n−−√n).

Tip: you might need to use the % operator that calculates a remainder from a division (*e.g.* 8 % 3 is 2).

The math module is already imported.

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Define the initial check: numbers lower than 2 are not prime.
* Define the loop checking if the number n is not prime.

import numpy as np

def is\_prime(n):

# Define the initial check

if n < 2:

return False

# Define the loop checking if a number is not prime

for i in range(2 , int(np.sqrt(n))):

if n % i != 0:

return False

return True

**Instructions 2/2**

**50 XP**

* [2](javascript:void(0))
* Filter prime numbers from cands into the primes list.

def is\_prime(n):

# Define the initial check

if n < 2:

return False

# Define the loop checking if a number is not prime

for i in range(2, int(math.sqrt(n)) + 1):

if n % i == 0:

return False

return True

# Filter prime numbers into the new list

primes = [num for num in cands if is\_prime(num)]

print("primes = " + str(primes))

cands = [1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49]

<script.py> output:

primes = [5, 13, 17, 29, 37, 41]

Great! Now you can do primarity checks for many more numbers! Remember, writing a function to check whether a number is prime is one of the most frequent interview questions – but it can have many variations.

**Exercise**

**Exercise**

**Coprime number sequence**

Two numbers aa and bb are coprime if their Greatest Common Divisor (GCD) is 1. GCD is the largest positive number that divides two given numbers aa and bb. For example, the numbers 7 and 9 are coprime because their GCD is 1.

Given two lists list1 and list2, your task is to create a new list coprimes that contains all the coprime pairs from list1 and list2.

But first, you need to write a function for the GCD using the following algorithm:

1. check if b=0b=0
   * if true, return aa as the GCD between aa and bb
   * if false, go to step 2
2. make a substitution a←ba←b and b←a%bb←a%b
3. go back to step 1

**Instructions 1/2**

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Define the while loop as described in the context.
* Complete the return statement.

def gcd(a, b):

# Define the while loop as described

while b != 0:

temp\_a = a

a = b

b = temp\_a % b

# Complete the return statement

return a

list1 = [5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70]

list2 = [7, 14, 21, 28, 35, 42, 49, 56, 63, 70, 77, 84, 91, 98]

**Instructions 2/2**

**50 XP**

* [2](javascript:void(0))
* Create a list of tuples defining pairs of coprime numbers from list1 and list2.

def gcd(a, b):

# Define the while loop as described

while b != 0:

temp\_a = a

a = b

b = temp\_a % b

# Complete the return statement

return a

# Create a list of tuples defining pairs of coprime numbers

coprimes = [(i, j) for i in list1

for j in list2 if gcd(i, j) == 1]

print(coprimes)

<script.py> output:

[(5, 7), (5, 14), (5, 21), (5, 28), (5, 42), (5, 49), (5, 56), (5, 63), (5, 77), (5, 84), (5, 91), (5, 98), (10, 7), (10, 21), (10, 49), (10, 63), (10, 77), (10, 91), (15, 7), (15, 14), (15, 28), (15, 49), (15, 56), (15, 77), (15, 91), (15, 98), (20, 7), (20, 21), (20, 49), (20, 63), (20, 77), (20, 91), (25, 7), (25, 14), (25, 21), (25, 28), (25, 42), (25, 49), (25, 56), (25, 63), (25, 77), (25, 84), (25, 91), (25, 98), (30, 7), (30, 49), (30, 77), (30, 91), (40, 7), (40, 21), (40, 49), (40, 63), (40, 77), (40, 91), (45, 7), (45, 14), (45, 28), (45, 49), (45, 56), (45, 77), (45, 91), (45, 98), (50, 7), (50, 21), (50, 49), (50, 63), (50, 77), (50, 91), (55, 7), (55, 14), (55, 21), (55, 28), (55, 42), (55, 49), (55, 56), (55, 63), (55, 84), (55, 91), (55, 98), (60, 7), (60, 49), (60, 77), (60, 91), (65, 7), (65, 14), (65, 21), (65, 28), (65, 42), (65, 49), (65, 56), (65, 63), (65, 77), (65, 84), (65, 98)]

Good job! Writing an algorithm to find the greatest common divisor is also one of the most popular coding interview questions. Now you know how to proceed! By the way, to impress interviewers, you can substitute lines 4-6 with just one line of code a, b = b, a % b.

# Combining iterable objects

You are given the list wlist that contains lists of different words. Your task is to create a new list of tuples, where each tuple contains a list from the wlist, its length, and the longest word. If there is ambiguity in choosing the longest word, the word with the lowest index in the considered list should be taken into account.

##### Instructions 1/3

**100 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* Define a function searching for the longest word given a list of words.

wlist = [['Python', 'creativity', 'universe'], ['interview', 'study', 'job', 'university', 'lecture'], ['task', 'objective', 'aim', 'subject', 'programming', 'test', 'research']]

wlist = [['Python', 'creativity', 'universe'], ['interview', 'study', 'job', 'university', 'lecture'], ['task', 'objective', 'aim', 'subject', 'programming', 'test', 'research']]

wlist = [['Python', 'creativity', 'universe'], ['interview', 'study', 'job', 'university', 'lecture'], ['task', 'objective', 'aim', 'subject', 'programming', 'test', 'research']]

# Define a function searching for the longest word

def get\_longest\_word(words):

longest\_word = ''

for word in words:

if len(word) > len(longest\_word):

longest\_word = word

return longest\_word

##### Instructions 2/3

**0 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* Create two lists: the lengths and longest words of each list in wlist.

# Define a function searching for the longest word

def get\_longest\_word(words):

longest\_word = ''

for word in words:

if len(word) > len(longest\_word):

longest\_word = word

return longest\_word

# Create lists with the lengths and longest words

lengths = [len(item) for item in wlist]

words = [zip(item) for item in wlist]

##### Instructions 3/3

**0 XP**

* [3](javascript:void(0))
* Combine wlist, lengths, and words into one iterable object and print each element.

# Define a function searching for the longest word

def get\_longest\_word(words):

longest\_word = ''

for word in words:

if len(word) > len(longest\_word):

longest\_word = word

return longest\_word

# Create lists with the lengths and longest words

lengths = [len(item) for item in wlist]

words = [get\_longest\_word(item) for item in wlist]

# Combine the resulting data into one iterable object

for item in zip(wlist , lengths , words):

print(item)

<script.py> output:

(['Python', 'creativity', 'universe'], 3, 'creativity')

(['interview', 'study', 'job', 'university', 'lecture'], 5, 'university')

(['task', 'objective', 'aim', 'subject', 'programming', 'test', 'research'], 7, 'programming')

Good job! Don't hesitate to play around with the zip object to fully understand its properties.

# Extracting tuples

In the previous exercise, you used two list comprehensions to create lists lengths and words that, respectively, refer to the lengths of the constituent lists in wlist and the longest words stored in those lists. In this exercise, you'll create them in a slightly different way. First, you'll need to put the same calculations into one list comprehension, which should result in a list of tuples. Second, apply the unzip operation to generate two distinct tuples, resembling lengths and words from the previous exercise.

The list wlist and the function get\_longest\_word() are already available in your workspace.

##### Instructions

**100 XP**

* Create a list of tuples each containing the length and the longest word of each item in wlist.
* Unwrap the created list and create two distinct tuples.

# Create a list of tuples with lengths and longest words

result = [

(len(item), get\_longest\_word(item)) for item in wlist

]

# Unzip the result

lengths, words = zip(\*result)

for item in zip(wlist, lengths, words):

print(item)

wlist = [['Python', 'creativity', 'universe'], ['interview', 'study', 'job', 'university', 'lecture'], ['task', 'objective', 'aim', 'subject', 'programming', 'test', 'research']]

<script.py> output:

(['Python', 'creativity', 'universe'], 3, 'creativity')

(['interview', 'study', 'job', 'university', 'lecture'], 5, 'university')

(['task', 'objective', 'aim', 'subject', 'programming', 'test', 'research'], 7, 'programming')

Very well! You're as master at zipping and unzipping!

# Creating a DataFrame

Your last task in this lesson is to create a DataFrame from a dictionary supplied by a zip object. You have to take each single word stored in the list wlist and calculate its length. This data should be stored in two separate tuples that are supplied to the zip() initializer. The resulting zip object should be used to construct a DataFrame where the first column will store words and the second column will store their lengths.

The module pandas is already imported for you as pd.

##### Instructions 1/4

**25 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Create a list with tuples where each has a word and the associated length.

# Create a list of tuples with words and their lengths

word\_lengths = [

(word , len(word)) for list\_1 in wlist for word in list\_1

]

##### Instructions 2/4

**25 XP**

* [2](javascript:void(0))
* [3](javascript:void(0))
* [4](javascript:void(0))
* Unwrap the word\_lengths and create two separate tuples words and lengths.

# Create a list of tuples with words and their lengths

word\_lengths = [

(item, len(item)) for items in wlist for item in items

]

# Unwrap the word\_lengths

words, lengths = zip(\*word\_lengths)

##### Instructions 3/4

**25 XP**

* [3](javascript:void(0))
* [4](javascript:void(0))
* Create a zip object combining column names for the future DataFrame and the associated data.

# Create a list of tuples with words and their lengths

word\_lengths = [

(item, len(item)) for items in wlist for item in items

]

# Unwrap the word\_lengths

words, lengths = zip(\*word\_lengths)

# Create a zip object

col\_names = ['word', 'length']

result = zip(col\_names , [words , lengths] )

##### Instructions 4/4

**25 XP**

* [4](javascript:void(0))
* Convert result to a dictionary and build a DataFrame.

# Create a list of tuples with words and their lengths

word\_lengths = [

(item, len(item)) for items in wlist for item in items

]

# Unwrap the word\_lengths

words, lengths = zip(\*word\_lengths)

# Create a zip object

col\_names = ['word', 'length']

result = zip(col\_names, [words, lengths])

# Convert the result to a dictionary and build a DataFrame

data\_frame = pd.DataFrame(dict(result))

print(data\_frame)

<script.py> output:

word length

0 Python 6

1 creativity 10

2 universe 8

3 interview 9

4 study 5

5 job 3

6 university 10

7 lecture 7

8 task 4

9 objective 9

10 aim 3

11 subject 7

12 programming 11

13 test 4

14 research 8

Great! You practiced the workflow to create a DataFrame from a zip object.

# Shift a string

You're going to create a generator that, given a string, produces a sequence of constituent characters shifted by a specified number of positions. For example, the string 'sushi' will result in the sequence 'h', 'i', 's', 'u', 's' if we use the shift of 2 positions to the right. If we use the shift of 2 positions to the left (or simply, -2), the resulting sequence will be 's', 'h', 'i', 's', 'u'.

Tip: the % operator might be helpful. Applying it to a positive or negative number gives a non-negative remainder (e.g. 8 % 3 = 2, -9 % 5 = 1)

##### Instructions

**100 XP**

* Define a for loop with the yield statement.
* Create a generator that shifts the string 'DataCamp' by 5 positions to the right.
* Create a new string using the generator and print it out.

def shift\_string(string, shift):

len\_string = len(string)

# Define a for loop with the yield statement

for idx in range(0, len\_string):

yield string[(idx - shift) % len\_string]

# Create a generator

gen = shift\_string('DataCamp', 5)

# Create a new string using the generator and print it out

string\_shifted = ''.join(gen)

print(string\_shifted)

<script.py> output:

aCampDat

Very well! As you might have noticed, this generator function can be used to shift any indexable object, not only strings.

# Throw a dice

Let's create an infinite generator! Your task is to define the simulate\_dice\_throws() generator. It generates the outcomes of a 6-sided dice tosses in the form of a dictionary out. Each key is a possible outcome (1, 2, 3, 4, 5, 6). Each value is a list: the first value is the amount of realizations of an outcome and the second, the ratio of realizations to the total number of tosses total.

Tip: use the randint() function from the random module (already imported). It generates a random integer in the specified interval (e.g. randint(1, 2) can be 1 or 2).

##### Instructions

**100 XP**

* Simulate a single toss to get a new number.
* Update the number and the ratio of realization.
* Yield the updated dictionary.
* Create the generator and simulate 1000 tosses.

def simulate\_dice\_throws():

total, out = 0, dict([(i, [0, 0]) for i in range(1, 7)])

while True:

# Simulate a single toss to get a new number

num = random.randint(1, 6)

total += 1

# Update the number and the ratio of realizations

out[num][0] = out[num][0] + 1

out[num][1] = round(out[num][0]/total, 2)

# Yield the updated dictionary

yield out

# Create the generator and simulate 1000 tosses

dice\_simulator = simulate\_dice\_throws()

for i in range(1, 1001):

print(str(i) + ': ' + str(next(dice\_simulator)))

Done it! As you can see from the output, the more you throw a dice, the more the ratios become equal. These values can be seen as probabilities of a single realization.

# Generator comprehensions

You are given the following generator functions:

def func1(n):

for i in range(0, n):

yield i\*\*2

def func2(n):

for i in range(0, n):

if i%2 == 0:

yield 2\*i

def func3(n, m):

for i in func1(n):

for j in func2(m):

yield ((i, j), i + j)

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))

Rewrite func1() as a generator comprehension with nn = 10.

# Rewrite func1() as a generator comprehension

gen = (i for i in func1(10))

for item in zip(gen, func1(10)):

print(item)

<script.py> output:

(0, 0)

(1, 1)

(4, 4)

(9, 9)

(16, 16)

(25, 25)

(36, 36)

(49, 49)

(64, 64)

(81, 81)

Rewrite func2() as a generator comprehension with nn = 20.

# Rewrite func2() as a generator comprehension

gen = (i for i in func2(20))

for item in zip(gen, func2(20)):

print(item)

<script.py> output:

(0, 0)

(4, 4)

(8, 8)

(12, 12)

(16, 16)

(20, 20)

(24, 24)

(28, 28)

(32, 32)

(36, 36)

Rewrite func3() as a generator comprehension with nn = 8 and mm = 10.

# Rewrite func3() as a generator comprehension

gen = (i for i in func3(8 , 10))

for item in zip(gen, func3(8, 10)):

print(item)

<script.py> output:

(((0, 0), 0), ((0, 0), 0))

(((0, 4), 4), ((0, 4), 4))

(((0, 8), 8), ((0, 8), 8))

(((0, 12), 12), ((0, 12), 12))

(((0, 16), 16), ((0, 16), 16))

(((1, 0), 1), ((1, 0), 1))

(((1, 4), 5), ((1, 4), 5))

(((1, 8), 9), ((1, 8), 9))

(((1, 12), 13), ((1, 12), 13))

(((1, 16), 17), ((1, 16), 17))

(((4, 0), 4), ((4, 0), 4))

(((4, 4), 8), ((4, 4), 8))

(((4, 8), 12), ((4, 8), 12))

(((4, 12), 16), ((4, 12), 16))

(((4, 16), 20), ((4, 16), 20))

(((9, 0), 9), ((9, 0), 9))

(((9, 4), 13), ((9, 4), 13))

(((9, 8), 17), ((9, 8), 17))

(((9, 12), 21), ((9, 12), 21))

(((9, 16), 25), ((9, 16), 25))

(((16, 0), 16), ((16, 0), 16))

(((16, 4), 20), ((16, 4), 20))

(((16, 8), 24), ((16, 8), 24))

(((16, 12), 28), ((16, 12), 28))

(((16, 16), 32), ((16, 16), 32))

(((25, 0), 25), ((25, 0), 25))

(((25, 4), 29), ((25, 4), 29))

(((25, 8), 33), ((25, 8), 33))

(((25, 12), 37), ((25, 12), 37))

(((25, 16), 41), ((25, 16), 41))

(((36, 0), 36), ((36, 0), 36))

(((36, 4), 40), ((36, 4), 40))

(((36, 8), 44), ((36, 8), 44))

(((36, 12), 48), ((36, 12), 48))

(((36, 16), 52), ((36, 16), 52))

(((49, 0), 49), ((49, 0), 49))

(((49, 4), 53), ((49, 4), 53))

(((49, 8), 57), ((49, 8), 57))

(((49, 12), 61), ((49, 12), 61))

(((49, 16), 65), ((49, 16), 65))

Nice work! Now you are able to answer any question on generators. Onwards to the next chapter where you will deep dive into functions!

# Positional arguments of variable size

Let's practice positional arguments of variable size. Your task is to define the function sort\_types(). It takes a variable number of positional arguments and checks if each argument is a number or a string. The checked item is inserted afterwards either in the nums or strings list. Eventually, the function returns a tuple containing these lists.

You can use the isinstance() function to check if an object is of a certain type (e.g. isinstance(1, int) returns True) or one of the types (e.g. isinstance(5.65, (int, str)) returns False).

Types to use in this task are int, float, and str.

##### Instructions

**100 XP**

* Define the function with an arbitrary number of arguments.
* Check if arg is a number and add it to nums if necessary.
* Check if arg is a string and add it to strings if necessary.

# Define the function with an arbitrary number of arguments

def sort\_types(\*args):

nums, strings = [], []

for arg in args:

# Check if 'arg' is a number and add it to 'nums'

if isinstance(arg, (int, float)):

nums.append(arg)

# Check if 'arg' is a string and add it to 'strings'

elif isinstance(arg, str):

strings.append(arg)

return (nums, strings)

print(sort\_types(1.0, 'car', 'hat', 4, 5, 'tree', 0.0))

<script.py> output:

([1.0, 4, 5, 0.0], ['car', 'hat', 'tree'])

Good! You can extend this code to sort many more data types.

# Keyword arguments of variable size

Now let's move to keyword arguments of variable size! Your task is to define the function key\_types(). It takes a variable number of keyword arguments and returns a new dictionary: the keys are unique object types of arguments passed to the key\_types() function and the associated values represent lists. Each list contains argument names that follow the type defined as a key (e.g. calling the key\_types(val1='a', val2='b', val3=1) results in {<class 'int'>: ['val3'], <class 'str'>: ['val1', 'val2']}).

To retrieve the type of an object, you need to use the type() function (e.g. type(1) is int).

##### Instructions

**100 XP**

* Define the function with an arbitrary number of keyword arguments.
* Iterate over key-value pairs.
* Update a list associated with a key.

<script.py> output:

{'a': 1, 'b': 2, 'c': (1, 2), 'd': 3.1, 'e': 4.2}

Nicely done! Now you practiced both positional and keyword arguments! Keep going!

# Combining argument types

Now you'll try to combine different argument types. Your task is to define the sort\_all\_types() function. It takes positional and keyword arguments of variable size, finds all the numbers and strings contained within them, and concatenates type-wise the results. Use the sort\_types() function you defined before (available in the workspace). It takes a positional argument of variable size and returns a tuple containing a list of numbers and a list of strings (type sort\_types? to get additional help).

##### Instructions

**100 XP**

* Define the arguments passed to the function (use any name you want).
* Find all the numbers and strings in the 1st argument.
* Find all the numbers and strings in the 2nd argument.

# Define the arguments passed to the function

def sort\_all\_types(\*args, \*\*kwargs):

# Find all the numbers and strings in the 1st argument

nums1, strings1 = sort\_types(\*args)

# Find all the numbers and strings in the 2nd argument

nums2, strings2 = sort\_types(\*kwargs.values())

return (nums1 + nums2, strings1 + strings2)

res = sort\_all\_types(

1, 2.0, 'dog', 5.1, num1 = 0.0, num2 = 5, str1 = 'cat'

)

print(res)

<script.py> output:

([1, 2.0, 5.1, 0.0, 5], ['dog', 'cat'])

Mastered! Arguments of variable size sometimes seem to be very confusing. But now you can handle them very well!

**Define lambda expressions**

Let's write some lambda expressions! You will be given three tasks: each will require you to define a lambda expression taking some values as arguments and using them to calculate a specific result.

**Instructions 1/3**

**35 XP**

* [1](javascript:void(0))

Take xx and return x2x2 if x>0x>0 and 00, otherwise.

# Take x and return x squared if x > 0 and 0, otherwise

squared\_no\_negatives = lambda x : x\*x if x >0 else 0

print(squared\_no\_negatives(2.0))

print(squared\_no\_negatives(-1))

Take a list of integers nums and leave only even numbers.

# Take a list of integers nums and leave only even numbers

get\_even = lambda nums: [n for n in nums if n % 2 == 0]

print(get\_even([1, 2, 3, 4, 5, 6, 7, 8, 9, 10]))

Take strings s1, s2 and list their common characters.

# Take strings s1, s2 and list their common characters

common\_chars = lambda s1 , s2 : [i for i in s1 for j in s2 if j == i]

print(common\_chars('pasta', 'pizza'))

<script.py> output:

4.0

0

<script.py> output:

[2, 4, 6, 8, 10]

<script.py> output:

['p', 'a', 'a']

Very good start on lambda expressions! As you can see, you can fold a pretty complicated code into a short statement and reuse it as much as you want.

# Converting functions to lambda expressions

Convert these three normally defined functions into lambda expressions:

# Returns a bigger of the two numbers

def func1(x, y):

if x >= y:

return x

return y

# Returns a dictionary counting charaters in a string

def func2(s):

d = dict()

for c in set(s):

d[c] = s.count(c)

return d

# Returns a squared root of a sum of squared numbers

def func3(\*nums):

squared\_nums = [n\*\*2 for n in nums]

sum\_squared\_nums = sum(squared\_nums)

return math.sqrt(sum\_squared\_nums)

##### Instructions 1/3

**35 XP**

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))

Convert func1() to a lambda expression.

# Convert func1() to a lambda expression

lambda1 = lambda x , y : x if x >= y else y

print(str(func1(5, 4)) + ', ' + str(lambda1(5, 4)))

print(str(func1(4, 5)) + ', ' + str(lambda1(4, 5)))

Convert func2() to a lambda expression.

# Convert func2() to a lambda expression

lambda2 = lambda s: dict([(c, s.count(c)) for c in set(s)])

print(func2('DataCamp'))

print(lambda2('DataCamp'))

Convert func3() to a lambda expression.

# Convert func3() to a lambda expression

lambda3 = lambda \*nums: math.sqrt(sum([n\*\*2 for n in nums]))

print(str(func3(3, 4)) + ', ' + str(lambda3(3, 4)))

print(str(func3(3, 4, 5)) + ', ' + str(lambda3(3, 4, 5)))

<script.py> output:

5, 5

5, 5

<script.py> output:

{'m': 1, 't': 1, 'C': 1, 'D': 1, 'p': 1, 'a': 3}

{'m': 1, 't': 1, 'C': 1, 'D': 1, 'p': 1, 'a': 3}

<script.py> output:

5.0, 5.0

7.0710678118654755, 7.0710678118654755

Good job! It is a very practical skill to understand when a normal function definition can be substituted with a lambda expression.

# Using a lambda expression as an argument

Let's pass lambda expressions as arguments to functions. You will deal with the list .sort() method. By default, it sorts numbers in increasing order. Characters and strings are sorted alphabetically. The method can be defined as .sort(key=function). Here, key defines a mapping of each item in the considered list to a sortable object (e.g. a number or a character). Thus, the items in a list are sorted the way sortable objects are.

Your task is to define different ways to sort the list words using the key argument with a lambda expression.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))

Sort words by string length.

# Sort words by the string length

words.sort(key=lambda s: len(s))

print(words)

<script.py> output:

['car', 'bag', 'job', 'time', 'cell', 'call', 'area', 'item', 'unit', 'truck', 'phone', 'shape', 'plane', 'leader', 'height', 'tequila', 'chicken', 'country', 'service', 'creature', 'interview', 'advantage', 'government', 'atmosphere', 'transaction']

Sort words by the last character in a string.

# Sort words by the last character in a string

words.sort(key=lambda s: s[-1])

print(words)

<script.py> output:

['tequila', 'area', 'job', 'time', 'service', 'phone', 'advantage', 'shape', 'atmosphere', 'creature', 'plane', 'bag', 'truck', 'cell', 'call', 'item', 'chicken', 'transaction', 'car', 'leader', 'government', 'height', 'unit', 'interview', 'country']

Sort words by the total amount of characters a, b, and c (e.g., the word 'cabana' has 3 a's, 1 b, and 1 c; in total, 5)

# Sort words by the total amount of certain characters

words.sort(key=lambda s: s.count('a') + s.count('b') + s.count('c'))

print(words)

<script.py> output:

['interview', 'time', 'government', 'phone', 'item', 'height', 'unit', 'truck', 'tequila', 'cell', 'leader', 'country', 'service', 'job', 'shape', 'atmosphere', 'plane', 'car', 'chicken', 'bag', 'call', 'area', 'creature', 'transaction', 'advantage']

Great work! In the next lesson you will cover more functions that use lambda expressions in their arguments.

# The map() function

Let's do some mapping!

Do you remember how zip() works? It merges given Iterables so that items with the same index fall into the same tuple. Moreover, the output is restricted by the shortest Iterable.

Your task is to define your own my\_zip() function with \*args depicting a variable number of Iterables. Rather than a zip object, my\_zip() should already return a list of tuples.

Comment: args should be checked whether they contain Iterables first. But we omit it for simplicity.

##### Instructions

**100 XP**

* Retrieve Iterable lengths from args using map() and find the minimal length.
* Append new items to the tuple\_list; each item is a list with elements from Iterables in args with the same index.

def my\_zip(\*args):

# Retrieve Iterable lengths and find the minimal length

lengths = list(map(len, args))

min\_length = min(lengths)

tuple\_list = []

for i in range(0, min\_length):

# Append new items to the 'tuple\_list'

tuple\_list.append(tuple(map(lambda x: x[i], args)))

return tuple\_list

result = my\_zip([1, 2, 3], ['a', 'b', 'c', 'd'], 'DataCamp')

print(result)

<script.py> output:

[(1, 'a', 'D'), (2, 'b', 'a'), (3, 'c', 't')]

Very good! Actually you could notice that sometimes map() can be substituted with a list comprehension. For example, list(map(lambda x: len(x), args)) can be re-written as [len(x) for x in args].

# The filter() function

Let's do some filtering! You will be given three corresponding tasks you have to complete. Use lambda expressions!

The variables strings and spells are available in your workspace.

##### Instructions 1/3

**50 XP**

* [1](javascript:void(0))

Filter out all the numbers in nums divisible by 3 or 5.

# Filter out all the numbers in nums divisible by 3 or 5

print(nums)

fnums = filter(lambda x: x % 3 != 0 and x % 5 != 0, nums)

print(list(fnums))

<script.py> output:

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

[1, 2, 4, 7, 8, 11, 13, 14, 16, 17, 19, 22, 23, 26, 28, 29, 31, 32, 34, 37, 38, 41, 43, 44, 46, 47, 49, 52, 53, 56, 58, 59, 61, 62, 64, 67, 68, 71, 73, 74, 76, 77, 79, 82, 83, 86, 88, 89, 91, 92, 94, 97, 98]

Return the string without its vowels ('y' is not a vowel in this case).

# Return the string without its vowels

print(string)

vowels = 'AEIOUaeiou'

fstring = filter(lambda x: x not in vowels, string)

print(''.join(fstring))

In [1]: string

Out[1]: 'Ordinary Least Squares'

<script.py> output:

Ordinary Least Squares

rdnry Lst Sqrs

Filter all the spells in spells with more than two 'a' characters.

# Filter all the spells in spells with more than two 'a's

print(spells)

fspells = filter(lambda x: x.count('a') > 2, spells)

print(list(fspells))

<script.py> output:

['riddikulus', 'obliviate', 'sectumsempra', 'avada kedavra', 'alohomora', 'lumos', 'expelliarmus', 'expecto patronum']

['avada kedavra']

Great! No surprise that we got 'avada kedavra' as the only Harry Potter spell with more than two as!

# The reduce() function

Now, it is time for some reduction! As before you'll be given three tasks to complete. Use lambda expressions!

The necessary functions from the functools module are already imported for you.

##### Instructions 1/3

**35 XP**

* [1](javascript:void(0))

Reverse a string using reduce().

# Reverse a string using reduce()

string = 'DataCamp'

inv\_string = reduce(lambda x, y: y + x, string)

print('Inverted string = ' + inv\_string)

<script.py> output:

Inverted string = pmaCataD

Find common items shared among all the lists in lists.

# Find common items shared among all the lists in lists

lists = [[1, 4, 8, 9], [2, 4, 6, 9, 10, 1], [9, 0, 1, 2, 4]]

common\_items = reduce(lambda x, y: set(x).intersection(y), lists)

print('common items = ' + str(common\_items))

<script.py> output:

common items = {9, 4, 1}

Convert a number sequence into a single number (e.g. [1, 2, 3] →→ 123).

# Convert a number sequence into a single number

nums = [5, 6, 0, 1]

num = reduce(lambda x, y: 10\*x + y, nums)

print(str(nums) + ' is converted to ' + str(num))

<script.py> output:

[5, 6, 0, 1] is converted to 5601

Nicely done! The functions map(), filter(), and reduce() are a serious weapon that you can use while solving a complex task. Now you are very well equiped!

# Calculate the number of function calls

Let's consider a classic example of recursion – the Fibonacci sequence, represented by non-negative integers starting from 0 with each element F(n)F(n) equals the sum of the preceding two: 0, 1, 1, 2, 3, 5, 8, 13, 21, .... You are given a function that returns a tuple with the nn-th element of the sequence and the amount of calls to fib() used:

def fib(n):

if n < 2:

return (n, 1)

fib1 = fib(n-1)

fib2 = fib(n-2)

return (fib1[0] + fib2[0], fib1[1] + fib2[1] + 1)

How many calls to fib() are needed to calculate the 15th15th and 20th20th elements of the sequence?

##### Instructions

**50 XP**

##### Possible Answers

* 

15th15th element: 3193 calls; 20th20th element: 35421 calls

* 

15th15th element: 1973 calls; 20th20th element: 21891 calls **(A)**

* 

15th15th element: 3193 calls; 20th20th element: 21891 calls

* 

15th15th element: 35421 calls; 20th20th element: 1973 calls

Correct! Notice how big the difference is in function calls, even though they are only 5 indices away from each other. Therefore, recursion has to be used with caution. Too many calls can lead to memory errors.

# Calculate an average value

We all know how to calculate an average value iteratively:

def average(nums):

result = 0

for num in nums:

result += num

return result/len(nums)

Could you provide a recursive solution? A formula for updating an average value given a new input might be handy:

x¯←xi+(n−1)x¯nx¯←xi+(n−1)x¯n

Here, x¯x¯ stands for an average value, xixi is a new supplied value which is used to update the average, and nn corresponds to the recursive call number (excluding the initial call to the function).

##### Instructions

**100 XP**

##### Instructions

**100 XP**

* Provide the base case for the algorithm.
* Define the recursive call for updating the average value.

# Calculate an average value of the sequence of numbers

def average(nums):

# Base case

if len(nums) == 1:

return nums[0]

# Recursive call

n = len(nums)

return (nums[0] + (n - 1) \* average(nums[1:])) / n

# Testing the function

print(average([1, 2, 3, 4, 5]))

Very good! Conversion of iterative algorithms to recursive ones and vice versa is a very frequent question in coding interviews.

<script.py> output:

3.0

Very good! Conversion of iterative algorithms to recursive ones and vice versa is a very frequent question in coding interviews.

# Approximate Pi with recursion

The number ππ can be computed by the following formula:

π=4∑k=0∞(−1)k2k+1=4(11−13+15−17+19−...)π=4∑k=0∞(−1)k2k+1=4(11−13+15−17+19−...)

Your task is to write a recursive function to approximate ππ using the formula defined above (the approximation means that instead of infinity ∞∞, the sequence considers only a certain amount of elements nn).

##### Instructions 1/2

**50 XP**

* [1](javascript:void(0))
* [2](javascript:void(0))
* Write a lambda expression calculating the kk-th element in the series (without taking 4 into account).

# Write an expression to get the k-th element of the series

get\_elmnt = lambda k: ((-1)\*\*k)/(2\*k+1)

##### Instructions 2/2

**50 XP**

* [2](javascript:void(0))
* Specify the base case.
* Define the recursive call (n is the number of elements to consider).

# Write an expression to get the k-th element of the series

get\_elmnt = lambda k: ((-1)\*\*k)/(2\*k+1)

def calc\_pi(n):

curr\_elmnt = get\_elmnt(n)

# Define the base case

if n == 0:

return 4

# Make the recursive call

return 4 \* curr\_elmnt + calc\_pi(n-1)

# Compare the approximated Pi value to the theoretical one

print("approx = {}, theor = {}".format(calc\_pi(500), math.pi))

<script.py> output:

approx = 3.143588659585789, theor = 3.141592653589793

Great job! Writing recursive functions for different mathematical series is a good exercise for understanding the concept of recursion. Keep practicing!