Assume that you already have a variable called 'x', which contains an integer value indicating the price of a company's stock. / Suppose that you have a put option on the stock with a strike price of 10. / The put option is just about to expire. Print its value.

```
print(max(10- x, 0))
"""
```

Assume that you already have a variable called 'x', which contains the percentage annual interest rate paid by a bank account as a nonnegative integer value. / Suppose that the balance of the bank account is now 100. / Print out what the balance of the bank account will be in 7 years. / Use the round function to round it to 3 digits.

```
print(round(100 * (1 + x / 100) ** 7, 3))
"""
```

Imagine that you are writing cashier software for Albert Heijn. The cashiers scan items one-by-one, and you have to take into account package discounts automatically / Assume that you already have a variable called 'package_price', another variable called 'individual_price' and a variable called 'package_size' indicating how many items a package contains. / You can assume that it is always cheaper to buy as many packages as possible to fill the order. Also, prices are integers. / Print the total amount to be paid when a customer buys 50 items. Make sure to print the integer value.

```
print((50 // package_size) * package_price + (50 % package_size) *
individual_price)
"""
```

Assume that you already have a variable called 'x' that contains a string value. / Print a string that equals 5 times the value in 'x', where every occurence of the value of 'x' is separated by the string '-5-'.

```
print(4 * (x + '-5-') + x)
"""
```

Assume that you already have a variable called 'x' that contains a boolean value. Print a string abc if 'x' is False and an empty string if 'x' is True.

```
print('abc' * (1 - x))
"""
```

Assume that you already have a variable called 'x', which contains a string value that represents an integer number. / Print out a float representation of the number that you get when you write '104' at the end of the value in 'x'.

```
"""
print(float(x + '104'))
"""
```

Assume that you already have one variable called 'x', which is a list that contains 4 elements. / Print a new list that contains elements of x in this order: first - third - fourth.

```
"""
print ([x[0], x[2], x[3]])
"""
```

Assume that you already have a variable called 'x', which contains a list with two elements. Each element is a further list with two elements: either 0 or 1. / Print a list that has the same structure as x, but all the 0's should be changed into 1's and all 1's into 0's.

```
"""
print([[1 - x[0][0], 1 - x[0][1]], [1 - x[1][0], 1 - x[1][1]]])
"""
```

Assume that you already have one variable called 'x', which is a list that contains at least 8 integer elements. Two elements of this list have the value 15. / Print a list with all the elements between the two 15's. / Both 15's should not be included. You can assume that there will only be two 15's in the list.

```
left = x.index(15) + 1
remainder_x = x[left:]
right = left + remainder_x.index(15)
print(x[left:right])
```

Week 2

Assume that you already have a variable called 'x', which contains a list. / The elements of the list are strings consisting of a single letter or a single digit. The list may contain duplicate elements. / Create a new variable 'y', which is a dictionary with each of the lowercase letters as keys and the frequency of each of those lowercase letters as corresponding values.

```
y = {}
for element in x:
  if element in "abcdefghijklmnopqrstuvwxyz":
    y[element] = x.count(element)
"""
```

Assume that you already have a variable called 'x', which contains an integer. / Create a new dictionary and call it 'y'. / Its keys should be integers indicating angles, measured in degrees, from 0 to 45 with a step size of 'x'. / The values should equal the cosine of the corresponding keys, rounded to 4 decimal digits. / Import the math package to calculate the cosine of a number.

```
"""
import math

y = {}
angle = 0
while angle <= 45:
  angle_in_radians = math.radians(angle)
  y[angle] = round(math.cos(angle_in_radians), 4)
  angle = angle + x
"""
```

Assume that you already have a variable called 'x', which contains a dictionary. / Modify 'x' with the dictionary y = {'b': 2, 'c': 2, 'd': 2} in the following way: - add to 'x' all key-value pairs of 'y' for which the key is not also present in 'x' / delete from 'x' all key-value pairs for which the key is also present in 'y'

```
"""
y = {'b': 2, 'c': 2, 'd': 2}
for key, value in y.items():
  if key in x:
    del x[key]
  else:
    x[key] = value
"""
```

Assume that there is already a variable 'x', which refers to a list with at least 5, but maybe more, different integers. / Create a dictionary called 'y', in which the keys are the indexes from the list
and the values are the values from the list, with one exception: for the highest value in the dictionary, the key should be "highest".

```
y = {}
for key, value in enumerate(x):
  y['highest' if value == max(x) else key] = value
"""
```

Assume that you already have two variables, which are two lists: one called 'keys', the other called 'values'. The lists have the same length. Both lists consists of integers. The 'keys' list contains
only unique numbers. / Create a dictionary 'x', for which all the keys are taken from the 'keys' list
and all the values from the 'values' list. Pairs are formed from elements with the same index in their lists. The only exceptions are when the values are divisible by 4, or when the key and the value
are equal to each other. In those cases, the key-value pair is ignored.
(below the zip pairs both elements!)

```
x = {}
for key, value in zip(keys, values):
  if key != value and value % 4 != 0:
    x[key] = value
"""
```

Assume that you already have a variable called 'x', which is a dictionary. / Print a list of the values of the dictionary, sorted in descending order of the corresponding dictionary keys.

```
sorted_keys = sorted(x.keys(), reverse=True)

result = []
for key in sorted_keys:
  result.append(x[key]) (this adds the value to the result list)

print(result)
"""
```

Assume that there are 2 dictionaries, called 'buyers' and 'sellers'. Both have 4 key-value pairs. Each value is a list with 3 integers. The values in the 'buyers' dictionary are the prices that each of the 4 buyers is willing to pay for the first, second, ... unit of a product in a market. / The values in the 'sellers' dictionary are the prices for which each of the 4 sellers is willing to sell the first, second, ... unit of the product in the market. / The product is homogeneous. If a seller's asking price is the same as a buyer's offer price, the buyer buys the item. / Print the integer value that indicates the number of items traded once all (welfare-increasing) transactions have taken place.

```
demand = []
for buyer_values in buyers.values():
  demand = demand + buyer_values
demand = sorted(demand, reverse=True)

supply = []
for seller_values in sellers.values():
  supply = supply + seller_values
supply = sorted(supply, reverse=False)

quantity_sold = 0
for buyer_value, supplyer_value in zip(demand, supply):
  quantity_sold += buyer_value >= supplyer_value

print(quantity_sold)
```

Write a function called 'main' that takes two arguments:
(1) a 5-character-long string made up of 3 symbols and 2 spaces in between them,
like this: "$ * #"
(2) a one-character optional keyword argument called 'losing_symbol' that has a default string value of '#'.
Your function should return another string, which can take one of two possible values.

```
def main(symbol_string, losing_symbol='#'):
  if symbol_string == f'{losing_symbol} {losing_symbol} {losing_symbol}':
    return f'{losing_symbol}-BLIMEY!!!'
  else:
    return '--'
"""
```

Write a function called 'main' that accepts two lists of integers as positional arguments. You can assume that the lists are equally long. / The function should return another list that has the same number of elements as either of the two input lists. In each position, the returned list should contain the smaller of the corresponding elements in the input lists. / If the corresponding input list elements happen to be equal, the returned list should contain the null value None in that position. / Consider using the 'zip' function for looping through the lists.

```
def main(list_1, list_2):
  result = []
  for a, b in zip(list_1, list_2):
    if a == b:
      result.append(None)
    else:
      result.append(min(a, b))
  return result
"""
```

Assume there is already a variable called 'integer_list' containing a list of integers. / Apply the lambda function lambda x: 4 * x + 5 to all elements of the list and print out the sum of the results as an integer.

```
print(sum(map(lambda x: 4 * x + 5, integer_list)))
```

Assume that you already have a variable called 'x', which is a list containing boolean values. / In this question, you are asked to carry out an operation on a list that is not implemented in Python by default, so you have to program it yourself. / The name of the operator is XNOR, which, when applied to a list of booleans, evaluates to True exactly when there is an odd number of True values in the list. / Print the result of applying XNOR to 'x'.

```
print(sum(x) % 2 == 1) (this is true when there is an odd number of true values)
"""
```

Assume that you already have a variable called 'x', which is a list containing integer values. Furthermore, you have 4 variables called 'a', 'b', 'c', 'd'. All of them contain an integer.
Print a list containing strings that describe the elements of 'x' as follows:
If the element is greater than the value of 'a', use "cat1". / If the element is less than the value of 'b', use "cat2". / If the element is greater than or equal to the value of 'c', use "cat3". / If the element is less than or equal to the value of 'd', use "cat4". / If the element fits into more than one category, use the highest category. / So, for example, if both "cat3" and "cat4" apply, use "cat4". / If the integer doesn't fall into any of the 4 categories, use "catunknown".

```
result = []
for element in x:
  if element <= d:
    text = "cat4"
  elif element >= c:
    text = "cat3"
  elif element < b:
    text = "cat2"
  elif element > a:
    text = "cat1"
  else:
    text = "catunknown"
  result.append(text)

print(result)
```

Week 3

```
"""
```

Write a function called 'main', that accepts two integer values, 'minimum' and 'maximum', as arguments. / The function should return a dictionary. The keys of the dictionary are tuples
for all possible combinations of integers between 'minimum' and 'maximum' (endpoints included). / The values of the dictionary are the result of the addition of the two integers in the corresponding key.

```
def main(minimum, maximum):
  result = {}
  for i in range(minimum, maximum + 1):
    for j in range(minimum, maximum + 1):
      result[(i, j)] = i + j
  return result
"""
```

Write a function called 'main' that accepts an arbitrary number of keyword arguments. You can assume that the value of every keyword argument is an integer. The function should return a dictionary. The keys of the dictionary should be the names of the keyword arguments. The values should be the remainders that you get when you divide the keyword argument value by 3. / Make sure that the dictionary values are integers. You may need to convert them.

```
def main(**kwargs):
  x = {}
  for key, value in kwargs.items():
    x[key] = int(value % 3)
  return x
"""
```

Write a function called 'main' that takes any number of positional arguments as inputs. In addition, it should also take an optional keyword argument that has a default integer value of 8. The keyword argument should be named 'forbidden_value'. / The positional arguments are all dictionaries. The keys and values of these dictionaries are all integers. Each dictionary key is unique across all the input dictionaries. / Your function should return a single dictionary that has all the key-value
pairs from every input dictionary, except for those pairs where the value equals the integer stored in 'forbidden_value'.

```
"""
def main(*args, forbidden_value=8):
  x = {}
  for dictionary in args:
    for key, value in dictionary.items():
      if value != forbidden_value:
        x[key] = value
  return x
"""
```

```
"""
```

Write a function called 'main' that takes two arguments: (1) a list of values that can be integers (int), floats (float), strings (str), or booleans (bool), (2) an optional keyword argument called 'excluded_type' that has a default value of int as a Python data type. / Your function should return another list that has all the elements of the input list (in the same order), except for those pairs where the type of the element equals the Python data type stored in 'excluded_type'.

```
def main(values, excluded_type=int):
  result = []
  for value in values:
    if type(value) != excluded_type:
      result.append(value)
  return result
"""
```

Write a function called 'main', that accepts an unlimited number of arguments. They are all integers and you can assume they are all different. The function should return a dictionary. The keys of the dictionary should be the arguments that are passed to the function. The values will be lists of
those passed arguments that are smaller than the key, and that the key can be divided by. The lists must be sorted in decreasing order.

```
def main(*args):
  x = {}
  for key in args:
    y = []
    for value in args:
      if value < key and key % value == 0:
        y.append(value)
    x[key] = sorted(y, reverse=True)
  return x
"""
```

Write a function called 'main' that accepts a list of integers. This is a list with the sales volumes of all the firms in a market. / The function should return the decrease in the Herfindahl-Hirschman Index (HHI) as an integer when a firm with a sales volume of 40 enters the market, while the sales volumes of the other firms stay the same.

```
def main(sales_before_entry):
  def calculate_hhi(sales):
    hhi = 0
    total = sum(sales)
    for s in sales:
      share = round((s / total) * 100)
      hhi += share ** 2
    return hhi

  hhi_before = calculate_hhi(sales_before_entry)
  hhi_after = calculate_hhi(sales_before_entry + [40])
  return hhi_before - hhi_after
"""
```

Types of objects
• Integer
var_1 = 123
print(type(var_1) == int)
• Floating point number (float)
var_1 = 123.3
• String (str)
var_1 = '123'
• Boolean (bool)
var_1 = True
• Tuple (Tuple)
var_1 = ('123', 123)
• List (list)
var_1 = [123, '123']
• Dictionary (dict)
var_1 = {'key_1': 'elem_1', 'key_2': 'elem_2'}
• Set (set)
var_1 = {123, '123'}

The Augmented Assignment Operators
• a = a + 1 -> a += 1
• a = a - 2 -> a -= 2
• a = a * 3 -> a *= 3
• a = a / 4 -> a /= 4
• a = a // 5 -> a //= 5
• a = a%6 -> a %= 6
• a = a ** 6 -> a **= 6

Arithmetic Operators
• a / b results in a float
• a // b results in the highest integer that is smaller or equal to the result of a division
• a % b gives the remainder of the integer division
• a ** b exponentiation

Boolean operators
• x and y: True if both are True, otherwise False
• x or y: False if both are False, otherwise True
• not x: True if x is False, False if X is True

Functions
• Functions can create objects, we say functions can return a value
def size(length, width):
return length * width
print(size (2,3)) # Onscreen: 6

Importing:
import numpy as np
print(np.random.randint(1, 10))
F-Strings:
a = 'Hello world'; b = 6; c = 0.0
print(f"We know {a =}, {b= }, and {c = }")

For-loops: examples
• You want to get the total of a series of numbers
• total = 0
for number in [1, 2, 3, 3, 5, 7]:
total += number
print (total) # Onscreen: 21
• You want to get the total of a series of numbers, but skip potential string values
• total = 0
for number in [1, 2, '3', 3, 5, 7]:
if type(number) == str:
continue
total += number
print (total) #Onscreen: 18
• **break:** skip to the first statement after the loop
• **continue:** go back to the statement with the **for** keyword

Loop over dictionaries: no sequences so needed to create sequences:
capitals.keys(): gives you a sequence of dictionary keys
capitals.values(): gives you a sequence of dictionary values
capitals.items(): gives you a sequence of tuples with for each pair the key and the value

Looping enumerate():
for index, country in enumerate(countries, 1):
print('country' + country + 'has index:' + str(index))
or: countries = ['Andorra', 'Belgium']
index = 1
for country in countries:
print(f'{country} has index: {index}')
index += 1
Onscreen: country Andorra has index: 1

Looping zip(): for 2 or more sequences/diff types
for country, capital in zip(countries, capitals):
print('country ' + country + ' has capital: ' + capital)
# Onscreen: country Andorra has capital: Andorra la Vella

Lambda functions
• Consider the following lambda function:
• add_two_numbers = lambda x,y: x + y
print(add_two_numbers(1,2))
• You could do the same as follows:
• def add_two_numbers (x,y):
return x+y
print(add_two_numbers(1,2))
• Lambda functions are useful in combination with the map function:
• l1 = [1, 2, 3, 4, 5, 6]
print(list(map(lambda x: x * 2, l1))) # Onscreen: [2, 4, 6, 8, 10, 12]
• With the filter function:
• l1 = [1, 2, 3, 4, 5, 6]
print(list(filter(lambda x: not x % 2, l1)))# Onscreen: [2, 4, 6]
• Note: you need the list function to convert a map or filter into a list, otherwise you get:
• print(map(lambda x: x * 2, l1)) # Onscreen:
• With the reduce function:
• from functools import reduce
l1 = [1, 2, 3, 4, 5, 6]
print(reduce(lambda x, y: x*y, l1)) # Onscreen: 720

Slicing (I)
• Slicing creates a new object, consisting of elements of another object
• Slicing can be used with objects of type list, string and tuple and the result will be a new object of the same type
• Syntax: **new_list = existing_list[start:end:step_size]**
• Start is inclusive, start the slicing. The default = 0
• **End is not inclusive** and tells Python before which index to stop the slicing
• **Step_size** is a parameter of which a positive sign tells you to look from left to right starting with the starting index, a negative value to look from right to left starting with the starting index. The absolute value tells you to skip that absolute value -1, for every step. The default is 1
- print(x[:2]) # star with first element {0 index}
- x[2:] # end with last element including last element
- x[:] # whole list
- To subset lists of lists, you can use the same technique as before:
- square brackets: x[2][0]

Mutable
• List (mu
\l1 = [1,2
print(l1) # Onscreen: [1, 2, 3]
l1[2] = 4
print(l1) # Onscreen: [1, 2, 4]
• Tuple (immutable):
t1 = (1,2,3)
print(t1) # Onscreen: (1, 2, 3)
t1[2] = 4
# Onscreen: TypeError Traceback (most recent call last)
Input In [2], in ()
----> 1 t1[2] = 4
TypeError: 'tuple' object does not support item assignment
Indexing
• print(l1[0] == l1[-len(l1)]) # True
• print(l1[-1] == l1[len(l1)]) # Error
• print(l1[-1] == l1[len(l1)-1]) # True
Last element on the list: len(existing_list) - 1
First element on the list: **-len(existing_list)**

Using dictionaries {key : value}
• How to create an empty dictionary:
• capitals = {}
• capitals = dict()
• How to create a filled dictionary:
• capitals = {'Andorra': 'Andorra la Vella', 'Belgium': 'Brussels'}
• countries = ['Andorra', 'Belgium']
capitals = ['Andorra la Vella', 'Brussels']
capitals = dict(zip(countries, capitals))
• How to select by key:
• print(capitals['Belgium'])
• How to delete a key:value pair:
• del(capitals['Belgium'])
• How to insert new key:value pairs:
• capitals['Netherlands'] = 'The Hague'
• capitals.update({'Netherlands' : 'The Hague', 'Belgium': 'Brussels'})
• How to update your dictionary:
• capitals['Netherlands'] = 'Amsterdam'
• capitals.update({'Netherlands' : 'Amsterdam', 'Belgium': 'Brussels'})
• Updating/inserting is a bit tricky: sometimes you must be sure that you are updating an existing key/value pair, and not inserting a new one, so:
• if 'Netherlands' in capitals:
capitals['Netherlands'] = 'The Hague'
else:
print("Country doesn't exist")
• Consider for example the following dictionary:
capitals = {'Andorra': 'Andorra la Vella', 'Belgium': 'Brussels'}
• capitals.keys() gives you a sequence of dictionary keys
• capitals.values() gives you a sequence of dictionary values
• capitals.items() gives you a sequence of tuples with for each pair the key and the value
• You can use the sequences mentioned on the previous slide in a for loop:
• capitals = {'Andorra': 'Andorra la Vella', 'Belgium': 'Brussels'}
• for key in capitals.keys():
print(key) # Onscreen: Andorra Belgium
• for key in capitals.values():
print(key) # Onscreen: Andorra la Vella Brussels
• for key, value in capitals.items():
print(key, value) # Onscreen: Andorra Andorra la Vella # Belgium Brussels
• The following lines of code do the same:
• for key in capitals:
• for key in capitals.keys():

• Assume that you already have one variable called 'x', which is a list that contains at least 8 integer elements. Two elements of this list have the value 17.
• Print a list with all the elements between the two 17's. Both 17's should not be included. You can assume that there will only be two 17's in the list.
• For example: • If x = [21, 17, 29, 20, 17, 21, 38, 26], your program should print: [29, 20]
• left = x.index(17) + 1
right = x.index(17, left)
print(x[left:right])
• print(x[(left:=x.index(17)+1): x.index(17,left)])
Database Conversions:
print('1' + str(1)) # Onscreen: 11
print(int('1') + 1) # Onscreen: 2
print(float('1.0') + 1) # Onscreen: 2.0
print(str(1232).count(str(2))) # Onscreen: 2

Slicing examples
example:
l1 = [1,2,3,4,5,6,7,8]
print(l1[5:1:-2]) # Onscreen: [6, 4]
Let's break it down:
start = 5 → Start at index 5, which is value 6
stop = 1 → Stop before index 1 (value 2) — so index 1 is not included
step = -2 → We're moving backwards, skipping every 2nd item
Step by step:
Start at index 5 → value 6
Move back by 2 → next index is 3 → value 4
Next would be index 1 — but we stop before we reach index 1
So we include:
l1[5] = 6
l1[3] = 4
And then we stop.
print(l1[5::-1]) # Onscreen: [6, 5, 4, 3, 2, 1]
print(l1[:1:-1]) # Onscreen: [8, 7, 6, 5, 4, 3]
print(l1[1:5:-2]) # Onscreen: []
s1 = 'uva Amsterdam'
print(s1[5:-3:2]) # Onscreen: mtr
Changing slice of the list:
l1 = [1, 2, 3, 4, 5, 6]
l1[1:4] = l1[1:4][::-1]
print(l1) # Onscreen: [1, 4, 3, 2, 5, 6]
Whats happening?
`l1[1:4] = [2, 3, 4]` (index 1 to 3)
`l1[1:4][::-1] = [4, 3, 2]` (reversed slice)
Tricky: l1 = [1, 2, 3, 4, 5, 6]
l1[1:2] = [1, 1]
print(l1) # Onscreen: [1, 1, 1, 3, 4, 5, 6]
`l1[1:2]` is `[2]` → we are replacing one item (at index 1)
But we're replacing it with **two items** → `[1,1]`
l1 = [1, 2, 3, 4, 5, 6]
l1[1] = [1, 1]
print(l1) # Onscreen: [1, [1, 1], 3, 4, 5, 6]
changing string:
s1 = 'uva Amxterdam' s1[6]= 's' #gives an error
s1 = s1[:6] + 's' + s1[7:]
print(s1)
changing tuple: t1 = (1, 4, 3) t1[1] = 2 # Gives an error
t1 = t1[:1] + (2,) + t1[2:]
print(t1)

Using lists index[value]
• How to create an empty list:
• countries = []
• countries = list()
• How to create a filled list:
• countries = ['Andorra', 'Belgium']
• How to select by index:
• print(countries[1])
• How to delete a value:
• del(countries[1])
• Question: How would you delete a value if you only knew the value
•
del(countries[countries.index('Andorra')])
add new value:countries += ['Netherlands','Belgium']
countries.append('Netherlands') → value that is not in the list or us intermediair
update list: countries[-1] = 'Holland'
Sets:collections with values {value1, value2, etc} , where each values is unique
countries.remove('Andorra')
countries.add('Andorra')
countries.update({'Andorra','Greece'}).pop (removes element from the list)

Function:
call a function:
def adder(argument1, argument2):
total = n1 + n2
return total
print(adder(1, 2)) # Onscreen: 3

Global and local names
• A global name is defined outside any function and is known everywhere, even inside functions, except when the same name is also defined inside a function, then the local name dominates the global name
• A global name can be defined via assignment:
• a = 1
• A local name is defined when a function is called, and is only known inside that function
• It's better to avoid the use of global, even though it is used a lot, and you need to know how it works
• Instead of:
• def changer():
global n1
n1 = n1 + 1
n1 = 1
changer()
print(n1) # Onscreen: 2
• Use:
• def changer(n1):
n1 = n1 + 1
return n1
a = 1
a = changer(a)
print(a) # Onscreen: 2
• So instead of using global we use the argument parameter mechanism for making data available to a function and return for making data available to the caller of a function21
Nested Functions: define a function inside another function
Keyword arguments: (doesn't have to be in order)
def calc(first, second, third, fourth, fifth, sixt):
return first + 2 * second + 3 * third + 4 * fourth \
+ 5 * fifth + 6 * sixt
****kwargs** → any number of keyword arguments in a function
all the same result:
adder(1, 2)
adder(1, y=2)/ adder(x = 1, 2) is wrong
adder(x=1,y=2)
Mutable Defaults:
def main (addition, l1 = []):
l1.append(addition)
return l1
print(main(2))
print(main(3)) #onscreen: [2] [2, 3]
→ python doesn't create new list