Documentation

-HINCU ALICE RAMONA-

*1.Problem statement*

The application must implement algorithms for:

* arithmetic operations for positive integers: addition, subtraction, multiplication and division by one digit, in a base p = {2,3,...,9,10,16}
* conversions of natural numbers between two bases p,q = {2,3,...,9,10,16} using the substitution method or successive divisions and rapid conversions between two bases p,q = {2, 4, 8, 16}.

and must have a menu such that all operations and conversion methods to be verified separately.

*2. Sub-algorithm’s diagram*

* **Start**
  + menu\_conversions
  + addition
  + subtraction
  + multiplication
  + division
  + input\_base\_and\_numbers
  + input\_and\_print\_menu\_addition
  + input\_and\_print\_menu\_subtraction
  + input\_and\_print\_menu\_division
  + input\_and\_print\_menu\_multiplication
* **addition**
* **subtraction**
* **multiplicaton**
* **division**
* **to\_ten**
* **from\_ten**
* **Menu\_conversions**
  + successive\_divisions
  + substitution\_method
  + intermediate\_base
  + rapid\_conversions
* **succesive\_divisions**
  + from\_ten
  + to\_ten
  + division
* **substitution\_method**
  + from\_ten
  + to\_ten
  + multiplication
  + addition
* **intermediate\_base**
  + succesive\_divisions
  + substitution\_method
* **transform\_digit\_to\_binary**
* **transform\_binary\_to\_digit**
* **convert\_to\_base**
  + transform\_binary\_to\_digit
* **covert\_to\_binary**
  + transform\_digit\_to\_binary
* **rapid\_conversions**
* **convert\_to\_base**
* **covert\_to\_binary**

*3. Used data type specification*

* list
* integer
* string

*4.Pseudo-code for algorithms and specification*

**4.1 Addition of two bases**

The pseudo-code is inspired from the seminar:

*c0=0; //c0, c1,… ,cn+1 Ԑ {0,1} are the carries used in addition  
for i=0,n  
 x’i(10) = xi(b) ; y’i(10) = yi(b) ; //convert the digits from base b in base 10  
 s10 = x’i(10) + y’i(10) + ci(10) ; //addition in base 10  
 s’i(10) = s10 mod b ; ci+1 = s10 div b;  
 si(b) = s’i(10); //convert the decimal value in base b  
end\_for  
si+1 = ci+1;*

The steps I am taking for my **addition** code:

* Fill with zeros the small number to make the numbers equal in length
* Parse through the numbers from right to left
* Convert the digits from both numbers in base 10
* Do the actual addition (with the carry) between the digits
* Update the carry and the final digit
* Make sure to add the last carry if it needs to
* We reverse the number and transform it into a string (from a list)

def addition(base, a, b):# Get the lengths of the 2 numbers  
 len\_a = len(a)  
 len\_b = len(b)  
 # The sums will be strings  
 s = ""  
 summ = 0  
 final\_number = []  
  
 # Fill with zeros the small number to make the numbers equal in length  
 diff = abs(len\_a - len\_b)  
 for i in range(1, diff + 1):  
 s += "0"  
 if len\_a < len\_b: # Condition to check if the strings have lengths mis-match  
 a = s + a  
 else:  
 b = s + b  
  
 # Start the algorithm  
 carry = 0  
 index = max(len\_a, len\_b) - 1 # we start from right to left  
 for i in range(index, -1, -1):  
 # We transform the digits in base 10  
 if a[i].isdigit():  
 digit\_a = ord(a[i]) - ord('0')  
 else: # in case it's a letter  
 digit\_a = ord(a[i]) - ord('0') - 7  
  
 if b[i].isdigit():  
 digit\_b = ord(b[i]) - ord('0')  
 else: # in case it's a letter  
 digit\_b = ord(b[i]) - ord('0') - 7  
  
 # Addition in base 10  
 summ = int(digit\_a) + int(digit\_b) + int(carry)  
 carry = summ // base  
 digit = summ % base  
  
 # we append to the list that represents the number  
 if digit > 9: # if the digit is a letter  
 final\_number.append(chr(digit + ord('0') + 7))  
 else:  
 final\_number.append(chr(digit + ord('0')))  
  
 if carry: # if we have a carry  
 final\_number.append(chr(carry + ord('0')))  
  
 # reverse the number  
 final\_number.reverse()  
 listToStr = ''.join(map(str, final\_number))  
 return listToStr

**4.2 Subtraction of two bases**

The pseudo-code is inspired from the seminar:

*c0=0; //c0, c1,… ,cn+1 Ԑ {0,1} are the borrows used in subtraction  
for i=0,n  
 x’i(10) = xi(b) ; y’i(10) = yi(b) ; //convert the digits from base b in base 10  
 d10 = x’i(10) - y’i(10) - ci(10) ; //subtraction in base 10  
 ci+1 = 0 ;  
 if (d10 < 0) then { d10 = d10+b ; ci+1 = 1;}  
 si(b) = s’i(10); //convert the decimal value in base b  
end\_for*

The steps I am taking for my **subtraction** code:

* Fill with zeros the small number to make the numbers equal in length
* Parse through the numbers from right to left
* Convert the digits from both numbers in base 10
* Do the actual subtraction (with the borrow) between the digits
* Update the borrow and the final digit
* Get rid of any 0s in the left
* We reverse the number and transform it into a string (from a list)

def subtraction(base, a, b):# Get the lengths of the 2 numbers  
 len\_a = len(a)  
 len\_b = len(b)  
 # The diff  
 s = ""  
 diff = 0  
 final\_number = []  
  
 # Fill with zeros the small number to make the numbers equal in length  
 diff\_abs = abs(len\_a - len\_b)  
 for i in range(1, diff\_abs + 1):  
 s += "0"  
 if len\_a < len\_b: # Condition to check if the strings have lengths mis-match  
 a = s + a  
 else:  
 b = s + b  
 # Start the algorithm  
 borrow = 0  
 index = max(len\_a, len\_b) - 1 # we start from right to left  
 for i in range(index, -1, -1):  
 # We transform the digits in base 10  
 if a[i].isdigit():  
 digit\_a = ord(a[i]) - ord('0')  
 else: # in case it's a letter  
 digit\_a = ord(a[i]) - ord('0') - 7  
  
 if b[i].isdigit():  
 digit\_b = ord(b[i]) - ord('0')  
 else: # in case it's a letter  
 digit\_b = ord(b[i]) - ord('0') - 7  
  
 # Subtraction in base 10  
 d = int(digit\_a) - int(digit\_b) - int(borrow)  
 borrow = 0  
 if d < 0:  
 d = d + base  
 borrow = 1  
 digit = d  
 # we append to the list that represents the number  
 if digit > 9: # if the digit is a letter  
 final\_number.append(chr(digit + ord('0') + 7))  
 else:  
 final\_number.append(chr(digit + ord('0')))  
  
 # In case we have 0s at the left of the number  
 while len(final\_number) != 1:  
 if final\_number[len(final\_number)-1] == '0':  
 final\_number.pop(len(final\_number)-1)  
 else:  
 break  
  
 final\_number.reverse()  
 listToStr = ''.join(map(str, final\_number))  
 return listToStr

**4.3 Multiplication between a number and a digit**

The pseudo-code is inspired from the seminar:

*c0=0; //c0, c1,… ,cn+1 Ԑ {0,1} are the carries used in addition  
y’(10) = y(b) ; //convert the digit from base b in base 10  
for i=0,n  
 x’i(10) = xi(b) ; //convert the digits from base b in base 10  
 p10 = x’i(10) \* y’(10) + ci(10) ; //multiplication in base 10  
 p’i(10) = p10 mod b ; ci+1 = p10 div b;  
 pi(b) = p’i(10); //convert the decimal value in base b  
end\_for  
pi+1 = ci+1;*

The steps I am taking for my **multiplication** code:

* I am swapping the numbers to make sure the second number is always a digit
* Convert the digit to base 10
* Parse through the numbers from right to left
* Convert the digits from the first number in base 10
* Do the actual multiplication (with the carry) between the digits
* Update the carry and the final digit
* Get rid of any 0s in the left
* We reverse the number and transform it into a string (from a list)

def multiplication(base, a, b):  
# Swap the two numbers in case a is not the number  
 if len(str(a)) < len(str(b)):  
 a, b = b, a  
 # In case it's not just one digit  
 if len(b) != 1:  
 raise ValueError("One number must have only 1 digit!")  
 # The product  
 product = 0  
 final\_number = []  
  
 # Start the algorithm  
 carry = 0  
 # Transform b in base 10  
 if b.isdigit():  
 digit\_b = ord(b) - ord('0')  
 else: # in case it's a letter  
 digit\_b = ord(b) - ord('0') - 7  
  
 index = len(a) - 1 # we start from right to left  
  
 for i in range(index, -1, -1):  
 # We transform the digit in base 10  
 if a[i].isdigit():  
 digit\_a = ord(a[i]) - ord('0')  
 else: # in case it's a letter  
 digit\_a = ord(a[i]) - ord('0') - 7  
  
 # Multiplication in base 10  
 product = int(digit\_a) \* int(digit\_b) + int(carry)  
 carry = product // base  
 digit = product % base  
  
 # we append to the list that represents the number  
 if digit > 9: # if the digit is a letter  
 final\_number.append(chr(digit + ord('0') + 7))  
 else:  
 final\_number.append(chr(digit + ord('0')))  
  
 if carry: # if we have a carry  
 final\_number.append(chr(carry + ord('0')))  
 final\_number.reverse()  
 listToStr = ''.join(map(str, final\_number))  
 return listToStr

**4.4. Division between a number and a digit**

The pseudo-code is inspired from the seminar:

*r = 0 //a decimal value  
y’(10) = y(b) ; //convert the digit from base b in base 10  
for i=n,0,-1  
 x’i(10) = xi(b) ; //convert the digits from base b in base 10  
 p10 = r \* b + x'i(10) ; //operations in base 10  
 q’i(10) = p10 mod b ; r = p10 div b;  
 qi(b) = q’i(10); //convert the decimal value in base b  
end\_for  
r(b) = r(10)*

The steps I am taking for my **division** code:

* I am swapping the numbers to make sure the second number is always a digit
* Convert the digit to base 10
* Parse through the numbers from left to right
* Convert the digits from the first number in base 10
* Do a multiplication (with the carry) between the digits, save the remainder and quotient
* Update the quotient and the remainder
* Get rid of any 0s in the left
* We reverse the number and transform it into a string (from a list)

def division(base, a, b):# Swap the two numbers in case a is not the number  
 if len(a) < len(b):  
 a, b = b, a  
 # In case it's not just one digit  
 if len(b) != 1:  
 raise ValueError("One number must have only 1 digit!")  
 # The quotient and remainder  
 q = 0  
 r = 0  
 final\_number = []  
  
 # Start the algorithm  
 # Transform b in base 10  
 if b.isdigit():  
 digit\_b = ord(b) - ord('0')  
 else: # in case it's a letter  
 digit\_b = ord(b) - ord('0') - 7  
  
 index = len(a) - 1 # we start from left to right  
  
 for i in range(0, index+1):  
 # We transform the digit in base 10  
 if a[i].isdigit():  
 digit\_a = ord(a[i]) - ord('0')  
 else: # in case it's a letter  
 digit\_a = ord(a[i]) - ord('0') - 7  
  
 # Operation in base 10  
 product = r \* base + digit\_a  
 q = product // digit\_b  
 r = product % digit\_b  
  
 # we append to the list that represents the number  
 if q > 9: # if the digit is a letter  
 final\_number.append(chr(q + ord('0') + 7))  
 else:  
 final\_number.append(chr(q + ord('0')))  
  
 if r > 9: # if the remainder is a letter  
 r = chr(r + ord('0') + 7)  
 else:  
 r = chr(r + ord('0'))  
  
 while len(final\_number) != 1:  
 if final\_number[0] == '0':  
 final\_number.pop(0)  
 else:  
 break  
  
 listToStr = ''.join(map(str, final\_number))  
 return r, listToStr

**4.5.0 Conversion from base 10 and to base 10**

Since the algorithms of division and multiplication are made for a number and a digit of the same base, we may have some problems with the conversions that involve the base 10. To avoid that, we just code some “mini-algorithm” for conversions from base 10 and to base 10.

The function named “from\_ten” has the role of converting a number from base 10 to a base b. It is a simplified “successive divisions” algorithm (since it’s from base 10), an algorithm that I will be talking about later (for pseudo-code, check 4.5.1)

def from\_ten(nr, base):  
 *"""  
 Converts given number nr, from base 10 to base b* ***:param*** *nr: the number that we need to convert(type: str)* ***:param*** *base: the base we convert the number (type:in)* ***:return*** *the converted number (type: str)  
 """* import string  
 r = ''  
 nr = int(nr)  
 while nr > 0:  
 # generate a string with numbers and letters, then select the digit that is the remainder  
 r = string.printable[nr % base] + r  
 nr //= base  
 return r

The function named “to\_ten” has the role of converting a number from base b to base 10. Using the int function, we can easily convert from any base to base 10.

def to\_ten(nr, b):  
 *"""  
 Converts given number nr, from base b to base 10  
 """* return int(nr, b)

Now let’s see the actual important algorithms.

**4.5.1 Successive divisions**

For the successive divisions, we can use the division algorithm that was already implemented and explained.

The pseudo-code :

*remainders = [] //a list where we save the remainders  
remainder ,quotient = division(b, xi(b), bf) // we do the division where b=initial base,*

*// bf = final base*

*remainders.append(remainder) //add remainder in the list*

*while quotient ≠0  
 remainder ,quotient = division(b, xi(b), bf)*

*remainders.append(remainder)   
end\_while  
reverse and transform in list*

The steps I am taking for my **successive divisions** code:

* the integer part is divided by the destination base obtaining a quotient and a remainder
* the quotient is divided by the destination base obtaining a new quotient and a new remainder
* ...
* -the process of successive divisions ends when 0 is obtained as a quotient
* -the remainders, in the reverse order of obtaining them, are the digits of the new representation in the destination base

def successive\_divisions(base\_start, nr, base\_end):# In case one of the bases are 10  
 if base\_start == 10:  
 return from\_ten(nr, base\_end)  
 if base\_end == 10:  
 return to\_ten(nr, base\_start)  
  
 if base\_start < base\_end:  
 raise ValueError("For substitution method, you should use a source base bigger than the destination base!")  
  
 # the remainders will be saved in a list  
 remainders = []  
  
 # we start the process of dividing  
 r, q = division(base\_start, nr, str(base\_end))  
 remainders.append(r)  
 while q != '0':  
 r, q = division(base\_start, q, str(base\_end))  
 remainders.append(r)  
  
 # reverse order of remainders  
 remainders.reverse()  
 listToStr = ''.join(map(str, remainders))  
 return listToStr

**4.5.2 Substitution method**

For the substitution method, we can use the multiplication and addition algorithms that were already implemented and explained.

The pseudo-code :

*products = [] //a list where we save the products  
power = 1 // the power of the base*

*for i=n,0,-1  
 p = multiplication(bf, xi(b), b)*

*power = multiplication(bf, b, power)*

*products.append(p)   
end\_for  
sum = products[0]*

*for i=0,n*

*sum = addition(bf, sum, products[i])*

*end\_for*

The steps I am taking for my **substitution method** code:

* all the digits from the source representation are converted into the destination base
* the base b is converted into the base h
* we use the formula: N'(h) = a'0(h) \* b'(h) ^ 0 + a'1(h) \* b'(h) ^ 1 + ... + a'm(h) \* b'(h) ^ m + a'(-1)(h) \* b'(h) ^ (-1) + ... + a'(-n)(h) \* b'(h) ^ (-n)
* since we only have natural numbers, we won't have negative powers

def substitution\_method(base\_start, nr, base\_end):  
# In case one of the bases are 10  
 if base\_start == 10:  
 return from\_ten(nr, base\_end)  
 if base\_end == 10:  
 return to\_ten(nr, base\_start)  
  
 if base\_start > base\_end:  
 raise ValueError("For substitution method, you should use a source base smaller than the destination base!")  
  
 # we start with the products  
 index = len(nr)-1  
  
 # we save the products in a list1  
 products = []  
 base\_start\_power = "1"  
 # we start from right to left  
 for i in range(index, -1, -1):  
 p = multiplication(base\_end, str(nr[i]), str(base\_start\_power))  
 base\_start\_power = multiplication(base\_end, str(base\_start), str(base\_start\_power))  
 products.append(p)  
  
 # we add the products  
 sum = products[0]  
 for i in range(1, len(products)):  
 sum = addition(base\_end, sum, products[i])  
 return sum

**4.5.2 Intermediate base**

For the intermediate base method, we can use the successive divisions and subtraction methods that were already implemented and explained.

The pseudo-code :

*dec = substitution\_method(b, n, 10) // convert n in decimal*

*fin = successive\_division(10, dec, bf) // convert in base bf*

The steps I am taking for my **intermediate base method** code:

* conversion from source base to base 10 (using the substitution method)
* conversion from base 10 into destination base (using the successive divisions method)

def intermediate\_base(base\_start, nr, base\_end):decimal\_number = substitution\_method(base\_start, nr, 10)  
 final\_number = successive\_divisions(10, decimal\_number, base\_end)  
 return final\_number

**4.5.2 Rapid conversions**

We use rapid conversions for bases that are powers of 2. For this algorithm, I chose to implement multiple functions so it would be easier to follow the process. I will only write the pseudo-code for the main algorithm and not for the functions, since this is what it matters, and just mention the other algorithms later (see 4.6)

The pseudo-code :

*number\_binary = n(2)*

*result = number\_binary(bf)*

The steps I am taking for my **substitution method** code:

From small to big:

* Each digit from the source number in base p=2^k will be replaced by the corresponding group of k binary digits (adding if it is necessary insignificant zeros to the left)

From big to small:

* for the integer part: from right to left make groups of k binary digits (eventually we add to the left insignificant zeros to have a complete group)
* the groups will be replaced by the corresponding digits in base q=2^k

def rapid\_conversions(base\_start, nr, base\_end):if base\_start not in [2, 4, 8, 16] or base\_end not in [2, 4, 8, 16]:  
 raise ValueError("For rapid conversions, the base should be equal to 2, 4, 8 or 16")  
  
 # the k binary digits  
 import math  
 exponent\_start = math.log2(base\_start)  
 exponent\_end = int(math.log2(base\_end))  
  
 # we convert the number to binary  
 if base\_start != 2:  
 number\_binary = convert\_to\_binary(nr, exponent\_start)  
 else:  
 number\_binary = nr  
  
 # we convert from binary to our final base  
 result = convert\_to\_base(number\_binary, exponent\_end)  
  
 return result

**4.6 Other algortihms**

**4.6.1. Transform binary to digit**

We transform a binary group to a digit

def transform\_binary\_to\_digit(group):digit = 0  
 power = 0  
 for i in range(len(group)-1, -1, -1):  
 digit += int(group[i]) \* (2 \*\* power)  
 power += 1  
  
 final\_digit = ""  
 if digit > 9: # if the digit is a letter  
 final\_digit = chr(digit + ord('0') + 7)  
 else:  
 final\_digit = chr(digit + ord('0'))  
 return str(final\_digit)

**4.6.2. Transform digit to binary**

We transform a digit to a binary group

def transform\_digit\_to\_binary(digit, exponent):# we start making the binary groups for every digit  
 binary\_group = []  
 while digit:  
 binary\_group.append(str(digit % 2))  
 digit //= 2  
  
 # fill with 0s where it's needed  
 while len(binary\_group) != exponent:  
 binary\_group.append("0")  
  
 # reverse it and make it a string  
 binary\_group.reverse()  
 listToStr = ''.join(map(str, binary\_group))  
 return listToStr

**4.6.3. Convert to binary**

Convert a number to base 2

def convert\_to\_binary(nr, exponent):number\_in\_binary = []  
 for i in range(len(nr)):  
 if nr[i].isdigit(): # digit in base 10  
 digit = ord(nr[i]) - ord('0')  
 else: # in case it's a letter  
 digit = ord(nr[i]) - ord('0') - 7  
  
 binary\_group = transform\_digit\_to\_binary(digit, exponent)  
  
 # put it in the final number which is a string  
 number\_in\_binary.append(binary\_group)  
  
 number\_final = ''.join(map(str, number\_in\_binary))  
 return number\_final

**4.6.4. Convert to base**

Convert a number from base 2 to a base b

def convert\_to\_base(number\_binary, exponent):exponent = 0 - exponent  
 result = ""  
  
 # we convert from binary to our final base  
 for i in range(len(number\_binary), -1, exponent):  
 binary\_group = number\_binary[(exponent + i):i]  
  
 # fill with 0s where we need  
 if i < (0-exponent):  
 if i == 0:  
 break  
 binary\_group = []  
 nr\_zero = 0 - exponent - i  
 print(nr\_zero)  
 while nr\_zero > 0:  
 binary\_group.append("0")  
 nr\_zero -= 1  
 for j in range(i):  
 binary\_group.append(number\_binary[j])  
 #i -= 1  
 listToStr = ''.join(map(str, binary\_group))  
 binary\_group = listToStr  
 digit = transform\_binary\_to\_digit(binary\_group)  
 result += digit  
  
 # reverse the string  
 result = result[::-1]  
 return result

**4.6.5. Input and check**

Two functions for input and checking a number or a base

**4.6.6. Input and print**

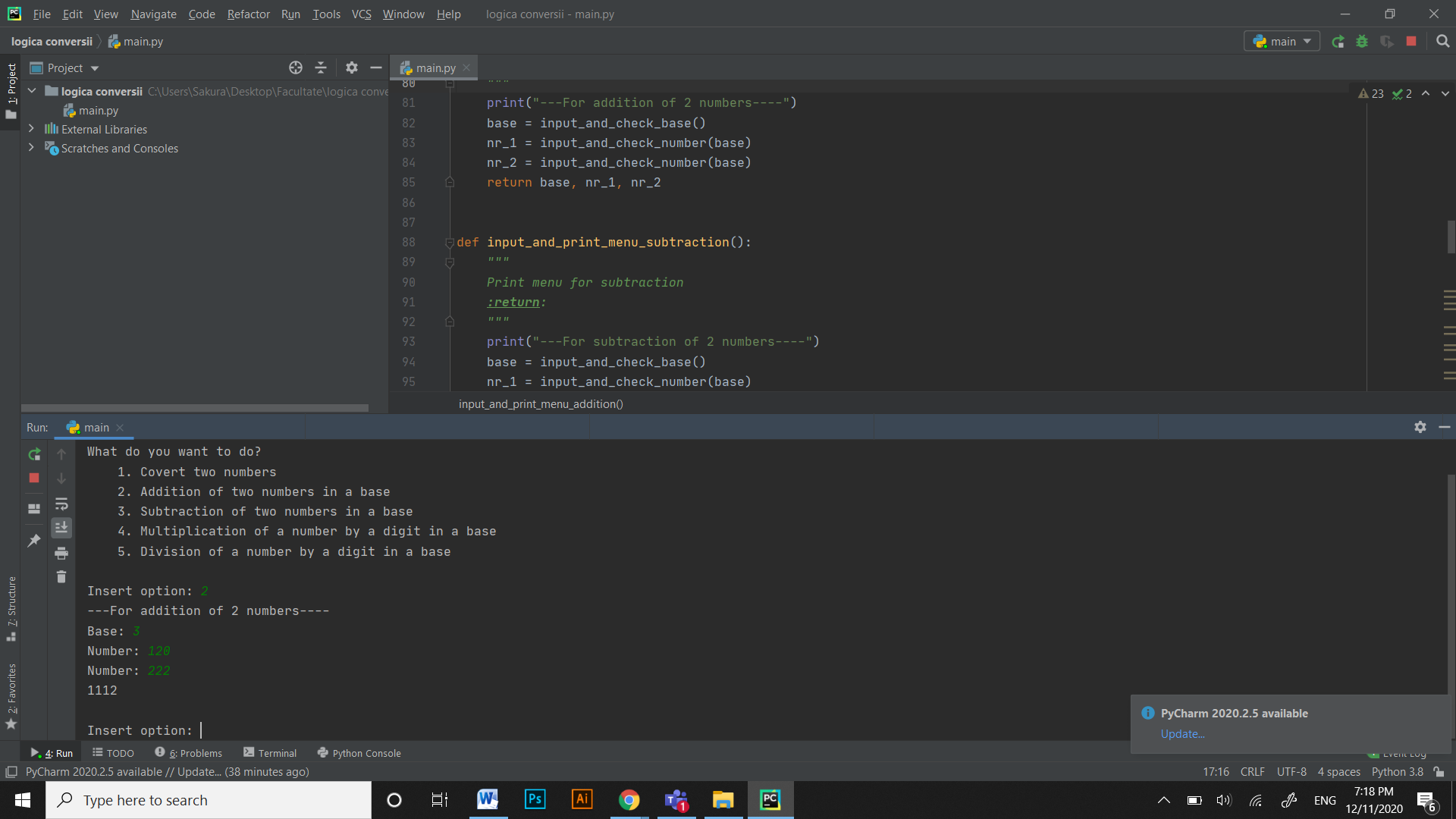
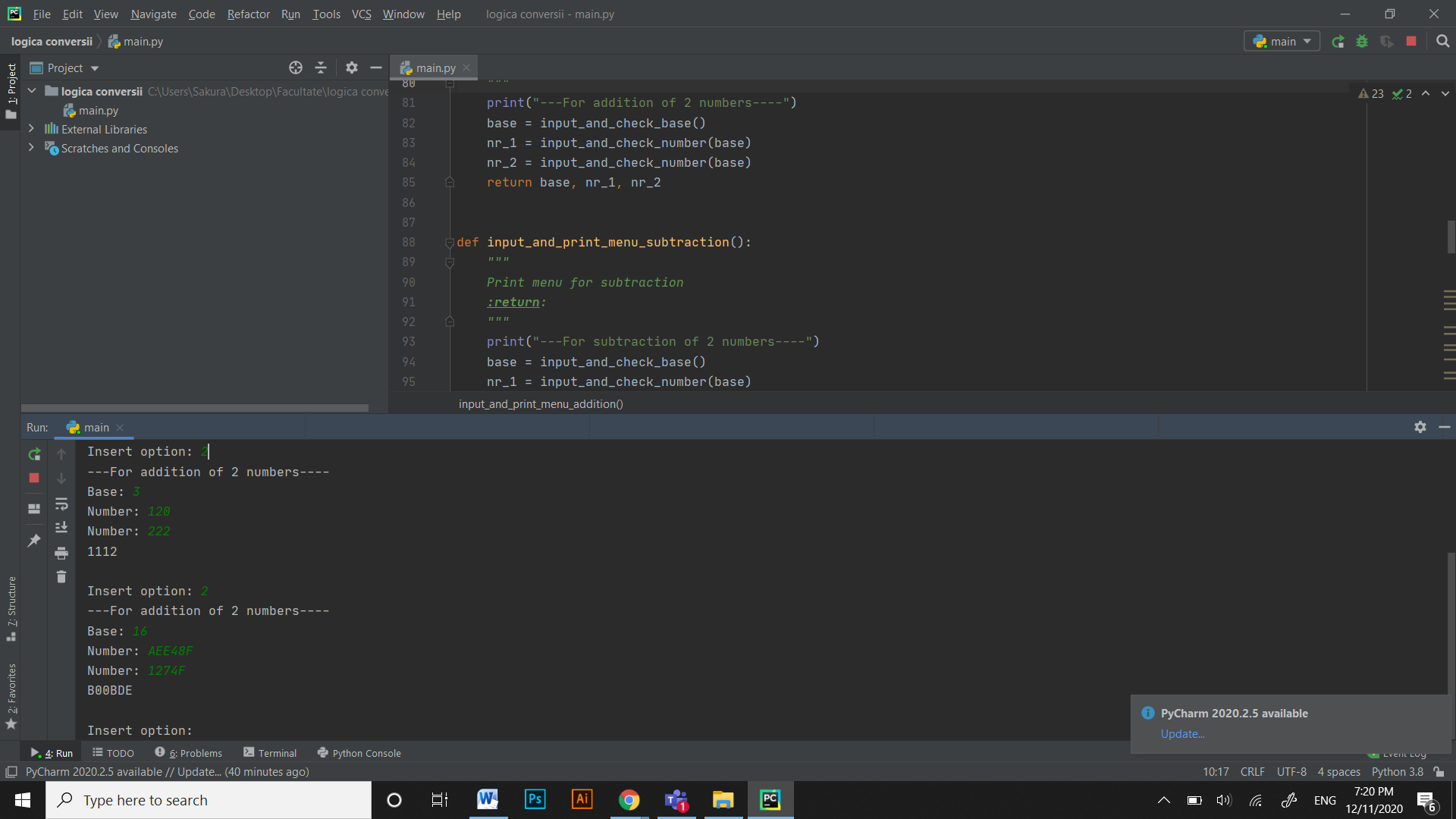
Multiple functions for menus

**4.6.7. Start**

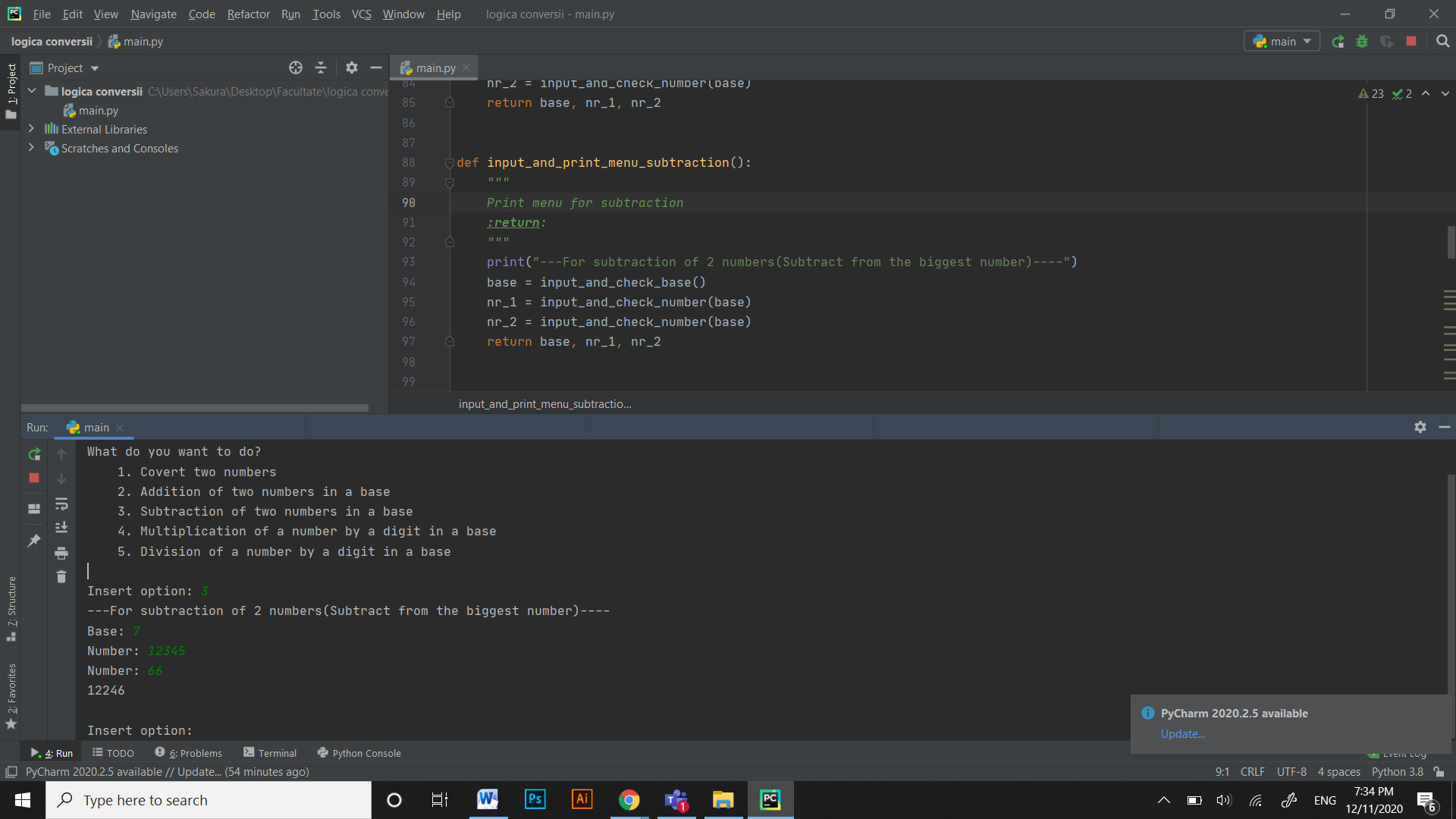
Here starts the program

*5.Test data*

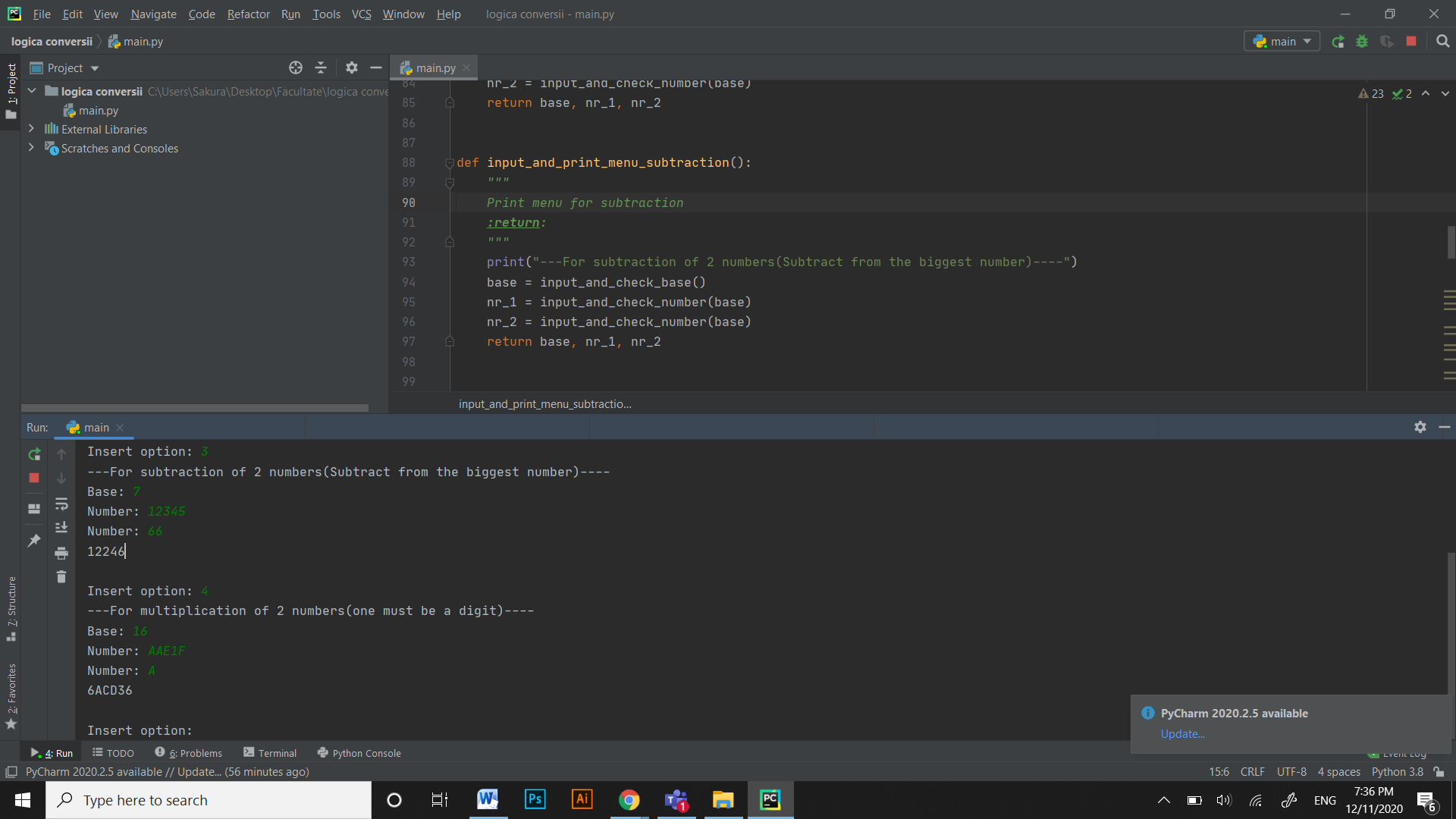
For addition

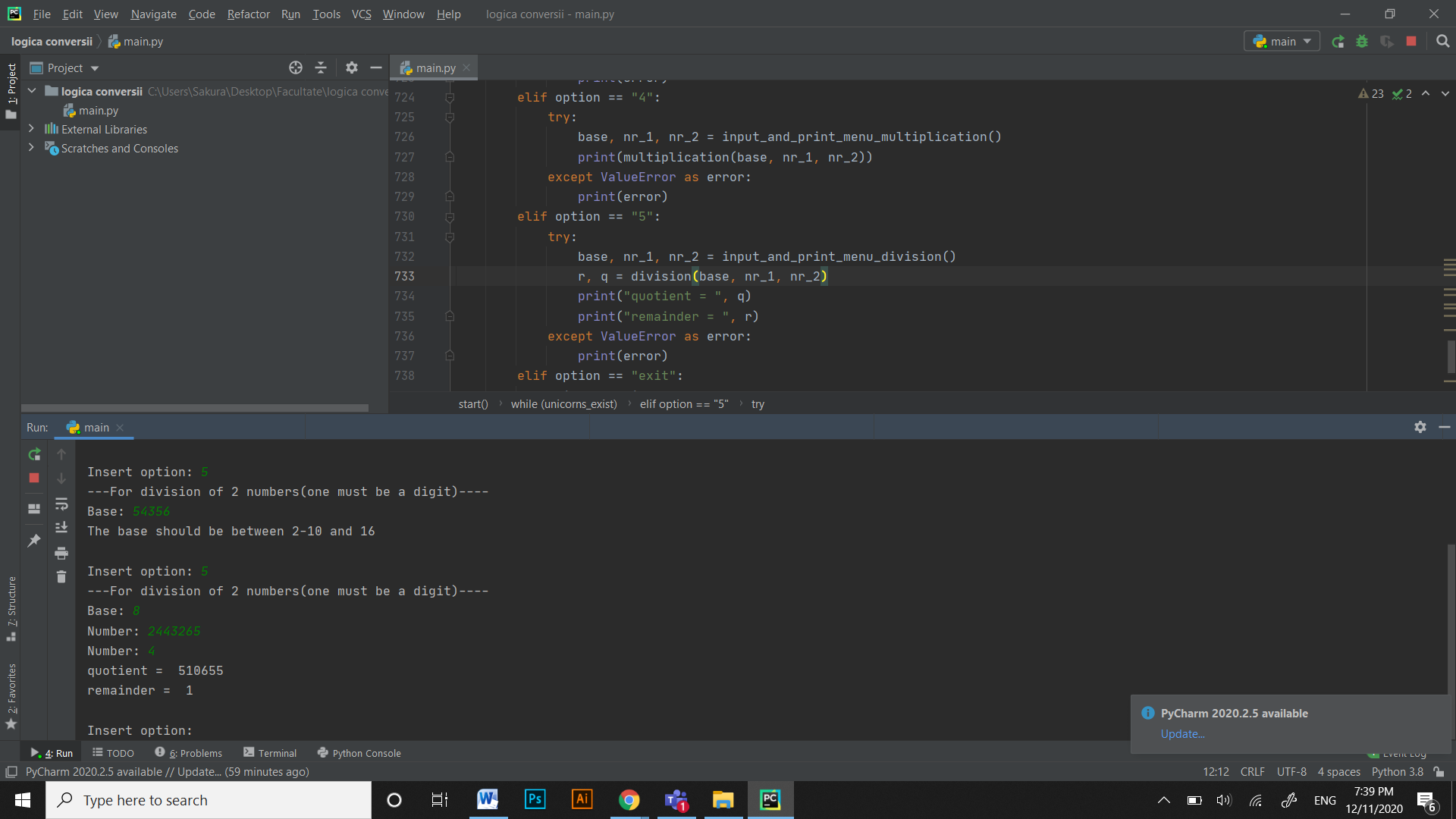
For subtraction



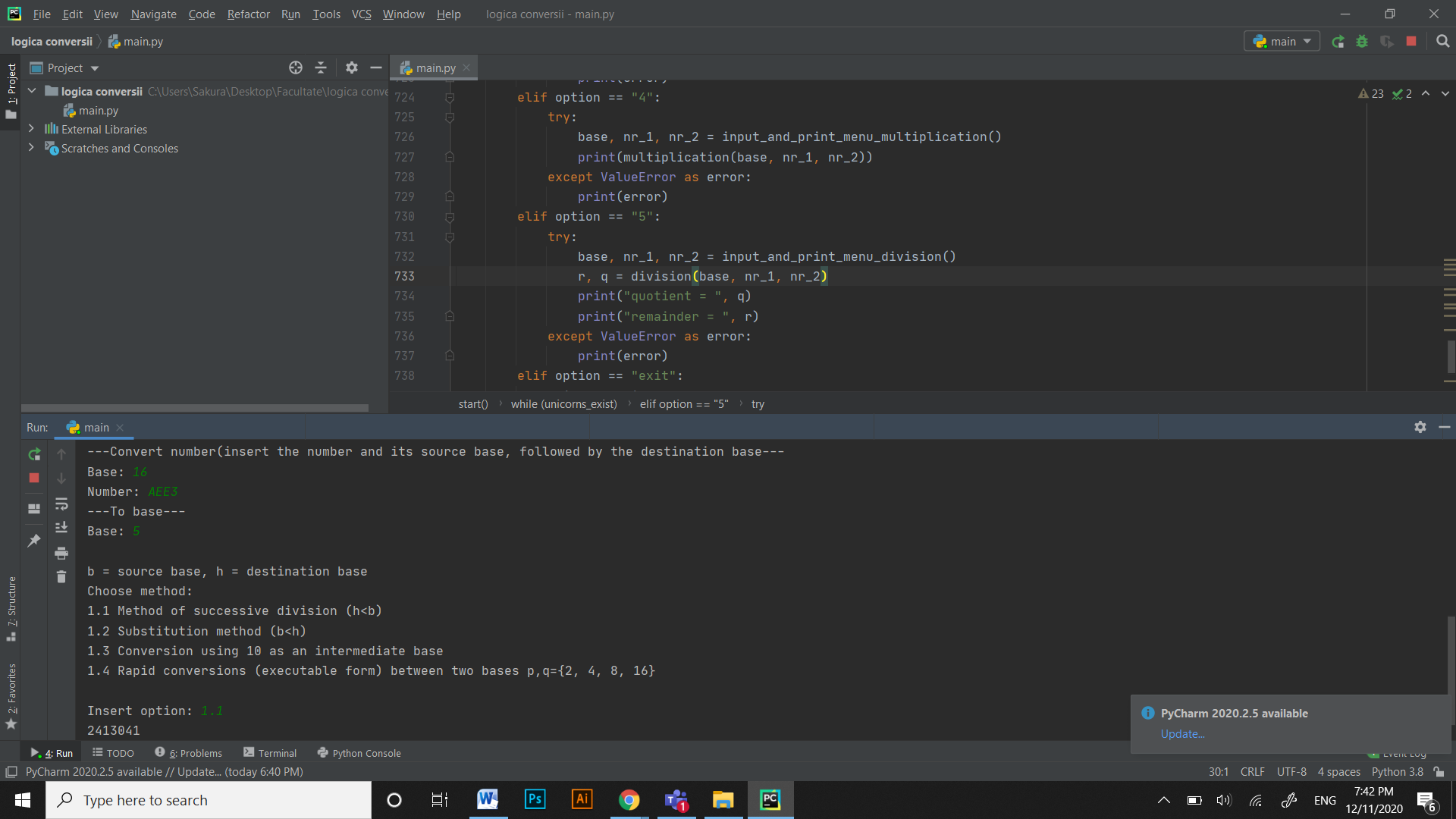
For multiplication



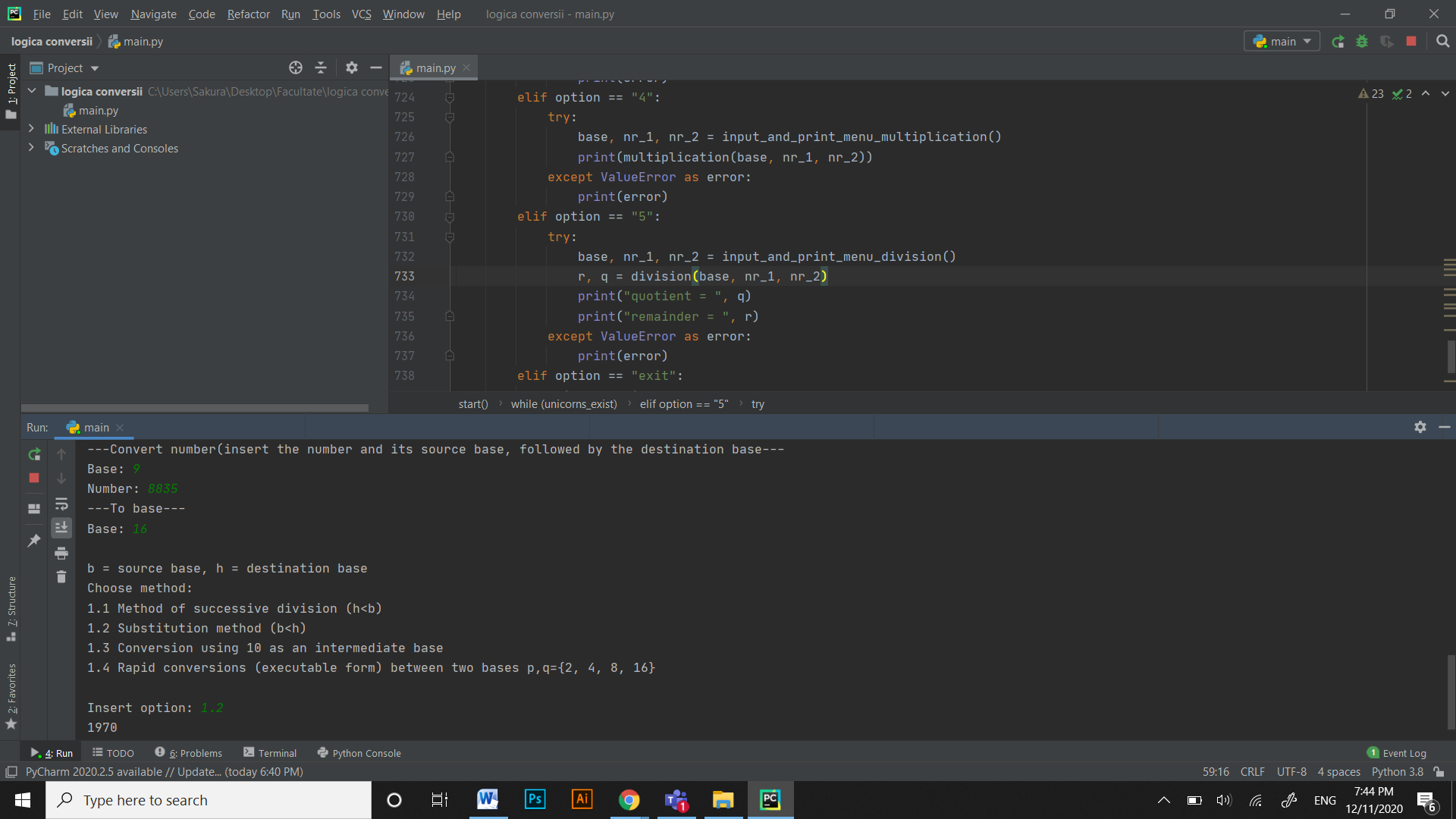
For division



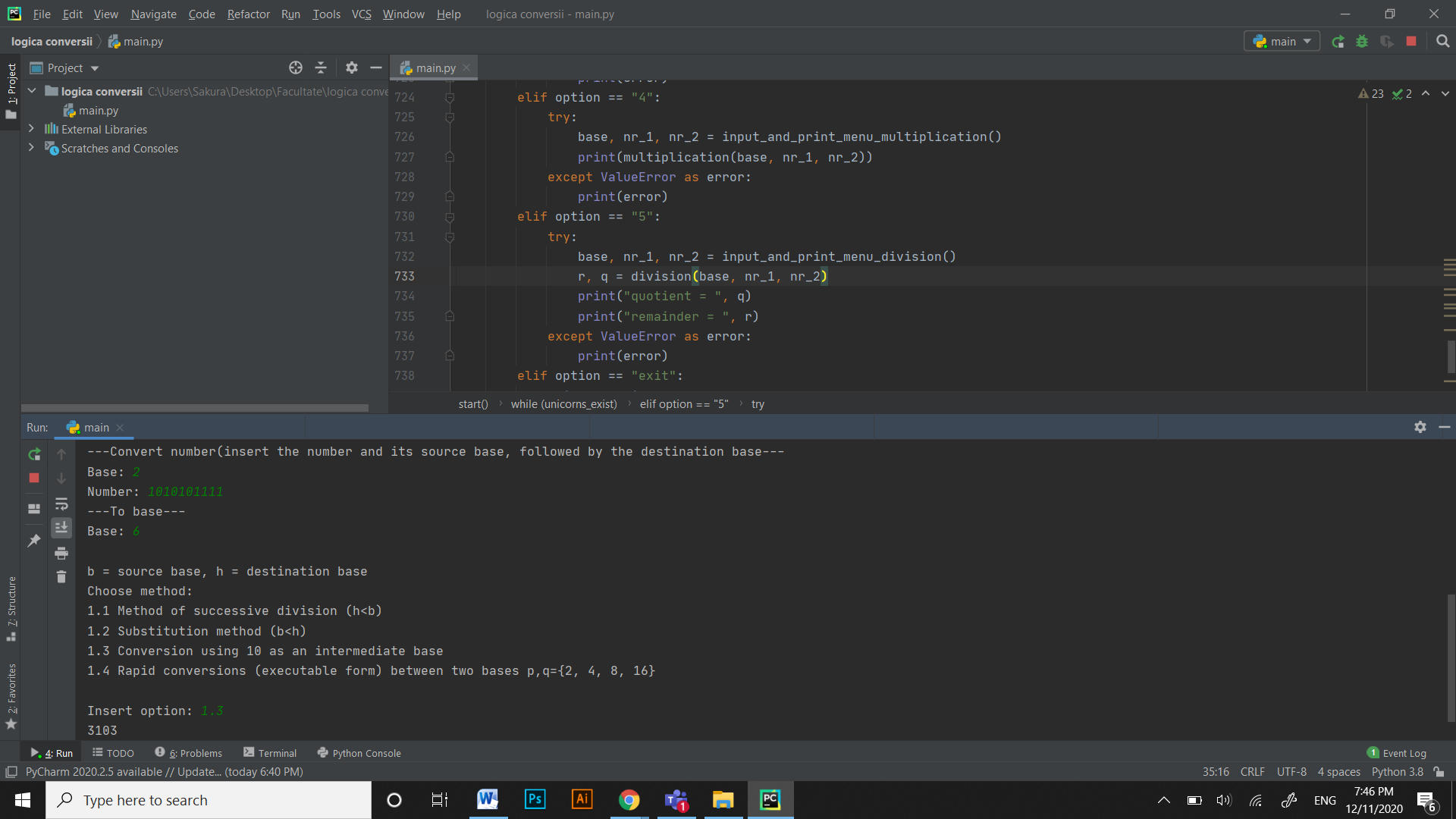
For successive divisions



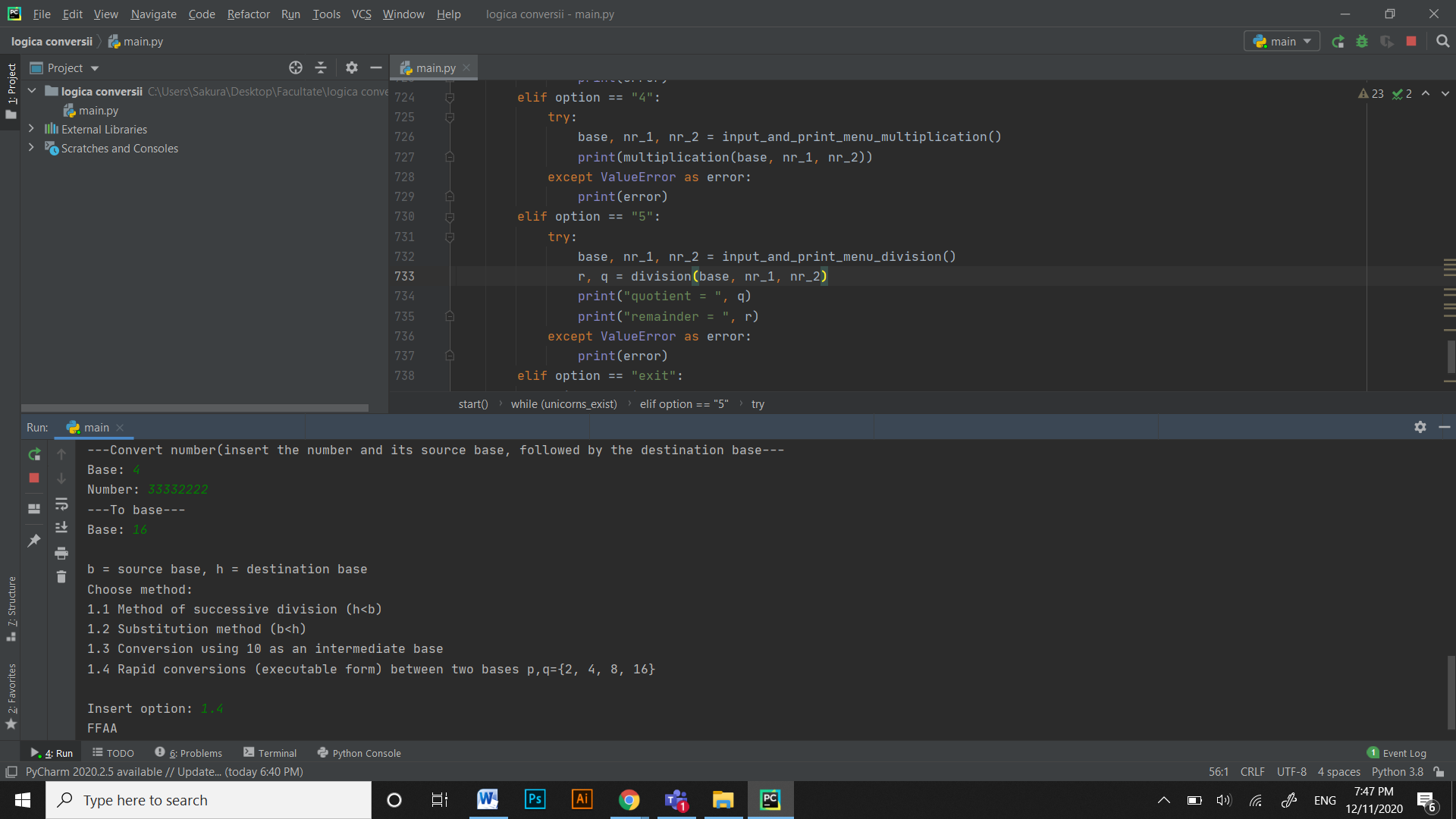
For substitution method



Conversion using 10 as an intermediate base



Rapid conversions



*6.Credits*

All the algorithms and functions are implemented by me(exception: from\_ten & to\_ten which are from the internet and modified a little)

The pseudo\_code and resources were taken from University of “Babeș-Bolyai”, Cluj, prof: Lupea Mihaiela.

If this project is uploaded to my github account, you are free to use any algorithm from here without giving me credit (for personal/commercial use), just don’t upload it somewhere else without the link of my github account(this is an exception).

Converter & calculator to verify:

https://www.rapidtables.com/convert/number/base-converter.html