Optional homework documentation

# 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, successive divisions or with 10 as an intermediary base 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.

# Sub-algorithm diagram:

start():

* add\_ui():
  + add(base, number1, number2):
    - \_\_add\_\_(other)
* sub\_ui():
  + sub(base, number1, number2):
    - \_\_sub\_\_(other)
* mul\_ui():
  + mul(base, number1, number2):
    - \_\_mul\_\_(other)
* div\_ui():
  + div(base, number1, number2):
    - \_\_truediv\_\_(other)
* conv\_div\_ui():
  + create(base, number)
  + successive\_div(number, base)
* conv\_sub\_ui():
  + create(base, number)
  + substitution(number, base):
    - successive\_div(number, base)
* conv\_inter\_ui():
  + creator(base, number)
  + intermediate\_base(number, base):
    - successive\_div(number, base)
    - substitution(number, base)
* conv\_rapid\_ui():
  + rapid\_convertions(number, base):
    - substitution(number, base)

# Used data types:

* int (integer) – used for the bases of the numbers and the calculations
* str (string) – used for the numbers read from the keyboard, being converted later on into other data types (int for their calculations and Number (object) for storing the numbers)
* Number (class object) – a data type created to store the actual number (str) and its base (int) and with methods which tell how to do each operation for any two numbers

# Specifications and pseudo-code:

1. Addition:  
     
   def \_\_add\_\_(self, other):  
    *'''  
    Adds two numbers together by adding each digit of each number from the right and keeping a carry if necessary* ***:param*** *other: the other number (Number object)* ***:return****: the result of the operation* ***:raises****: NumberError if the 'other' parameter is not of type Number or if the numbers don't have the same base  
    '''* if type(other) != type(self):  
    raise NumberError('Cannot add a non number to a number')  
    if self.base != other.base:  
    raise NumberError('Cannot add two numbers of different bases')  
    result = ''  
    carry = 0  
    number1 = self.number[::-1]  
    number2 = other.number[::-1]  
    for index in range(max(len(number1), len(number2))):

# loops trough the numbers from right to left and adds the digits of each number with the carry, which is set afterwards to 0 or 1, and puts the result into the result string  
 digit\_sum = carry  
 if len(number1) <= index:  
 pass  
 elif '0' <= number1[index] <= '9':  
 digit\_sum += int(number1[index])  
 else:  
 digit\_sum += (ord(number1[index])-55)  
 if len(number2) <= index:  
 pass  
 elif '0' <= number2[index] <= '9':  
 digit\_sum += int(number2[index])  
 else:  
 digit\_sum += (ord(number2[index])-55)  
 carry = int(digit\_sum/self.base)  
 digit = digit\_sum%self.base  
 if 10 <= digit:  
 digit = chr(digit+55)  
 result += str(digit)

# if the last carry is 1 than we put it to the right of the result (the result will be reversed when returned, since all the calculations were done from right to left)  
 if carry != 0:  
 result += str(carry)  
 return Number(self.base, result[::-1])

1. Subtraction:

def \_\_sub\_\_(self, other):

'''

Subtracts two numbers by subtracting each digit of each number from the right and keeping a remainder if necessary

:param other: the other number (Number object)

:return: the result of the operation

:raises: Number error if the 'other' parameter is not of type Number, if the numbers don't have the same base

or if the second number is larger than the first one

'''

if type(other) != type(self):

raise NumberError('Cannot subtract a non number to a number')

if self.base != other.base:

raise NumberError('Cannot subtract two numbers of different bases')

if self < other:

raise NumberError('Cannot subtract a smaller number from a larger number')

result = ''

transport = 0

number1 = self.number[::-1]

number2 = other.number[::-1]

for index in range(len(number1)):

# loops trough the numbers from right to left, subtracting the digitof the second number from the digit of the firs number and adds the transport, which is then set to 0 or -1 accordingly, and saves the operation in the result string

digit\_dif = 0

if '0' <= number1[index] <= '9':

digit\_dif += int(number1[index])

else:

digit\_dif += (ord(number1[index]) - 55)

if index >= len(number2):

digit\_dif -= 0

elif '0' <= number2[index] <= '9':

digit\_dif -= int(number2[index])

else:

digit\_dif -= (ord(number2[index]) - 55)

digit\_dif += transport

# if the operation’s result is < 0, the transport takes the value of -1 and the base of the numbers is added to the result

if digit\_dif < 0:

transport = -1

digit\_dif += self.base

else:

transport = 0

if digit\_dif <= 9:

result += str(digit\_dif)

else:

result += chr(digit\_dif + 55)

index = 0

# since the operations were done from right to left we have to reverse the result

result = result[::-1]

# we eliminate the unnecessary 0-s at the beginning of the number

while index < len(result) - 1 and result[index] == '0':

result = result[index+1:]

return Number(self.\_base, result)

1. Multiplication:

def \_\_mul\_\_(self, other):  
 *'''  
 Multiplies two numbers together by multiplying each digit of the first number from the right with the second  
 number and keeps a carry if necessary* ***:param*** *other: the other number (Number object)* ***:return****: the result of the operation* ***:raises****: NumberError if the 'other' parameter is not of type Number, if the numbers don't have the same base  
 or if the 'other' number has more than one digit  
 '''* if type(other) != type(self):  
 raise NumberError('Cannot multiply a non number to a number')  
 if self.base != other.base:  
 raise NumberError('Cannot multiply two numbers of different bases')  
 if len(other.number) != 1:  
 raise NumberError('Cannot multiply by more than one digit')  
 result = ''  
 carry = 0  
 number1 = self.number[::-1]  
 number2 = other.number  
 if 'A' <= number2:  
 number2 = (ord(number2) - 55)  
 else:  
 number2 = int(number2)  
 for index in range(len(number1)):

# loops trough each digit of the first number and multiplies it by the second number, adding the carry to it, which is set to the integer division between the product of the digits and the base of the numbers  
 digit\_product = 1  
 if '0' <= number1[index] <= '9':  
 digit\_product \*= int(number1[index])  
 else:  
 digit\_product \*= (ord(number1[index])-55)  
 digit\_product \*= number2  
 digit\_product += carry  
 carry = int(digit\_product/self.base)  
 digit = digit\_product%self.base  
 if 10 <= digit:  
 digit = chr(digit+55)  
 result += str(digit)

# if the carry is not 0 we add it to the end of the result (the result will be reversed when returned, since all the calculations were done from right to left)  
 if carry != 0:  
 if 10 <= carry:  
 carry = chr(carry+55)  
 result += str(carry)  
 return Number(self.base, result[::-1])

1. Division:  
     
   def \_\_truediv\_\_(self, other):  
    *'''  
    Divides two numbers by taking each digit from the first number from left and divide it by the second number,  
    keeping a remainder if necessary* ***:param*** *other: the other number (Number object)* ***:return****: the result of the operation* ***:raises****: Number error if the 'other' parameter is not of type Number, if the numbers don't have the same base,  
    if the second number has more than one digit or if the second number is 0  
    '''* if type(other) != type(self):  
    raise NumberError('Cannot divide a non number to a number')  
    if self.base != other.base:  
    raise NumberError('Cannot divide two numbers of different bases')  
    if len(other.number) != 1:  
    raise NumberError('Cannot divide by more than one digit')  
    if other.number == '0':  
    raise NumberError('Cannot divide by 0')  
    quotient = ''  
    index = 0  
    dividend = self.number[index]

# we take the dividends from left to right from the number; if at first one digit is not large enough for the division, then we will take the first 2 digits of the number so we do not have unnecessary 0-s in the beginning  
 if dividend < other.number and index + 1 < len(self):  
 index += 1  
 dividend += self.number[index]  
 divisor = (int(other.number) if '0' <= other.number <= '9' else (ord(other.number) - 55))  
 while index < len(self.number):

# we loop trough the digits of the number and add them at the end of the resulted dividend, then divide the resulted number by the divisor, adding the result into the quotient and putting the remainder into the new dividend  
 if len(dividend) <= 1:  
 d = (int(dividend) if '0' <= dividend <= '9' else (ord(dividend) - 55))  
 else:  
 d = (int(dividend[1]) if '0' <= dividend[1] <= '9' else (ord(dividend[1]) - 55))  
 d += self.base \* (int(dividend[0]) if '0' <= dividend[0] <= '9' else (ord(dividend[0]) - 55))  
 q = d // divisor  
 r = d % divisor  
 quotient += (str(q) if 0 <= q <= 9 else chr(q + 55))  
 dividend = (str(r) if 0 <= r <= 9 else chr(r + 55))

# if the number does not have any more digits to take from, then we break out of the loop  
 if index + 1 < len(self):  
 index += 1  
 dividend += self.number[index]  
 else:  
 break

# the remainder is equal to the last dividend calculated  
 remainder = dividend  
 return Number(self.base, quotient), remainder

1. Conversion: Successive divisions:  
     
   def successive\_div(self, number, base2):  
    *'''  
    Converts a number to the destination base using successive divisions* ***:param*** *number: the number to convert (Number object)* ***:param*** *base2: the base to convert the number to (int)* ***:return****: the converted number  
    '''*

# if we convert from base 10 we do simple divisions by the destination base adding the remainders into the result string and the divide the remainder until it is 0 if number.base == 10:  
 result = ''  
 number\_int = int(number.number)  
 while number\_int != 0:  
 quotient = number\_int // base2  
 remainder = number\_int % base2  
 if remainder <= 9:  
 result += str(remainder)  
 else:  
 result += chr(remainder + 55)  
 number\_int = quotient

# the result string will be reversed since we need to take the remainders from right to left  
 return Number(base2, result[::-1])  
 else:

# for a base different than 10, we do the same thing except we have to apply the division rule from the division algorithm to the quotient until it is 0, adding the last remainder into the result string every time we divide it  
 result = ''  
 number\_str = number.number  
 base2\_str = (str(base2) if base2 <= 9 else chr(base2 + 55))  
 while number\_str != '0':  
 quotient = ''  
 index = 0  
 dividend = number\_str[index]  
 if dividend < base2\_str and index + 1 < len(number\_str):  
 index += 1  
 dividend += number\_str[index]  
 divisor = base2  
 while index < len(number\_str):  
 if len(dividend) <= 1:  
 d = (int(dividend) if '0' <= dividend <= '9' else (ord(dividend) - 55))  
 else:  
 d = (int(dividend[1]) if '0' <= dividend[1] <= '9' else (ord(dividend[1]) - 55))  
 d += number.base \* (int(dividend[0]) if '0' <= dividend[0] <= '9' else (ord(dividend[0]) - 55))  
 q = d // divisor  
 r = d % divisor  
 quotient += (str(q) if 0 <= q <= 9 else chr(q + 55))  
 dividend = (str(r) if 0 <= r <= 9 else chr(r + 55))  
 if index + 1 < len(number\_str):  
 index += 1  
 dividend += number\_str[index]  
 else:  
 break  
 remainder = dividend  
 result += remainder  
 number\_str = quotient

# since we have to reverse the result string, we will eliminate the unnecessary 0-s at the end of it then revert it  
 while result[len(result) - 1] == '0':  
 result = result[:-1]  
 return Number(base2, result[::-1])

1. Conversions: Intermediary base 10:  
     
   def intermediate\_base(self, number, base2):  
    *'''  
    Converts a number to the destination base using an intermediary base (10)* ***:param*** *number: the number to convert (Number object)* ***:param*** *base2: the base to convert the number to (int)* ***:return****: the converted number  
    '''*

# converts the number from its base to base 10 using the substitution method, then it converts it into the destination base using the successive divisions methodnumber\_base10 = self.substitution(number, 10)  
 number\_destination\_base = self.successive\_div(number\_base10, base2)  
 return number\_destination\_base

1. Conversions: Substitution:  
     
   def substitution(self, number, base2):  
    *'''  
    Converts a number to the destination base using the substitution method* ***:param*** *number: the number to convert (Number object)* ***:param*** *base2: the base to convert the number to (int)* ***:return****: the converted number  
    '''*

# if the destination base is 10, it does all the operations in base 10; it multiplies each digit of the number to the number’s base at the power of the digit’s position, then it adds the results togetherif base2 == 10:  
 result = 0  
 number\_inv = number.number[::-1]  
 for poz in range(len(number)):  
 if '0' <= number\_inv[poz] <= '9':  
 digit = int(number\_inv[poz])  
 else:  
 digit = ord(number\_inv[poz]) - 55  
 result += digit \* pow(number.base, poz)  
 result = str(result)  
 return Number(base2, result)  
 else:

# if the destination base it’s not 10, the operations are done in the destination base; it converts, using successive divisions, the result of digit \* base^pozition for each digit, and adds the number to the result  
 result = Number(base2, '0')  
 number\_inv = number.number[::-1]  
 for poz in range(len(number)):  
 if '0' <= number\_inv[poz] <= '9':  
 digit = int(number\_inv[poz])  
 else:  
 digit = ord(number\_inv[poz]) - 55  
 result += self.successive\_div(Number(10, str(digit \* pow(number.base, poz))), base2)  
 return result

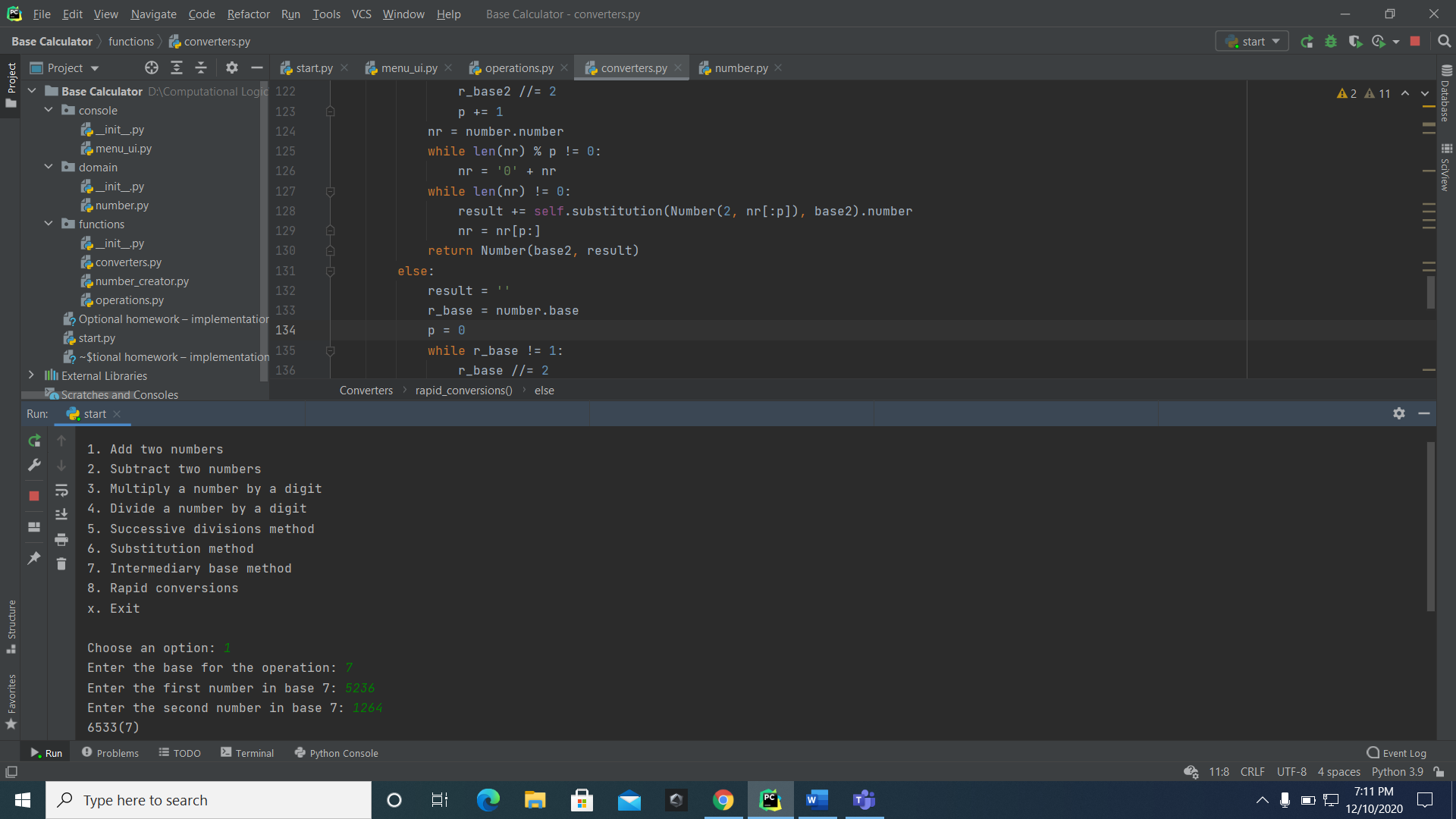
1. Conversions: Rapid conversions:  
     
   def rapid\_conversions(self, number, base2):  
    *'''  
    Converts a number to the destination base using rapid conversions* ***:param*** *number: the number to convert (Number object)* ***:param*** *base2: the base to convert the number to (int)* ***:return****: the converted number  
    '''* if number.base == 2 and base2 not in [4, 8, 16]:  
    raise ConversionError('Rapid conversions can only be done from base 2 into bases 4, 8, 16 and vice versa')  
    if base2 == 2 and number.base not in [4, 8, 16]:  
    raise ConversionError('Rapid conversions can only be done from base 2 into bases 4, 8, 16 and vice versa')

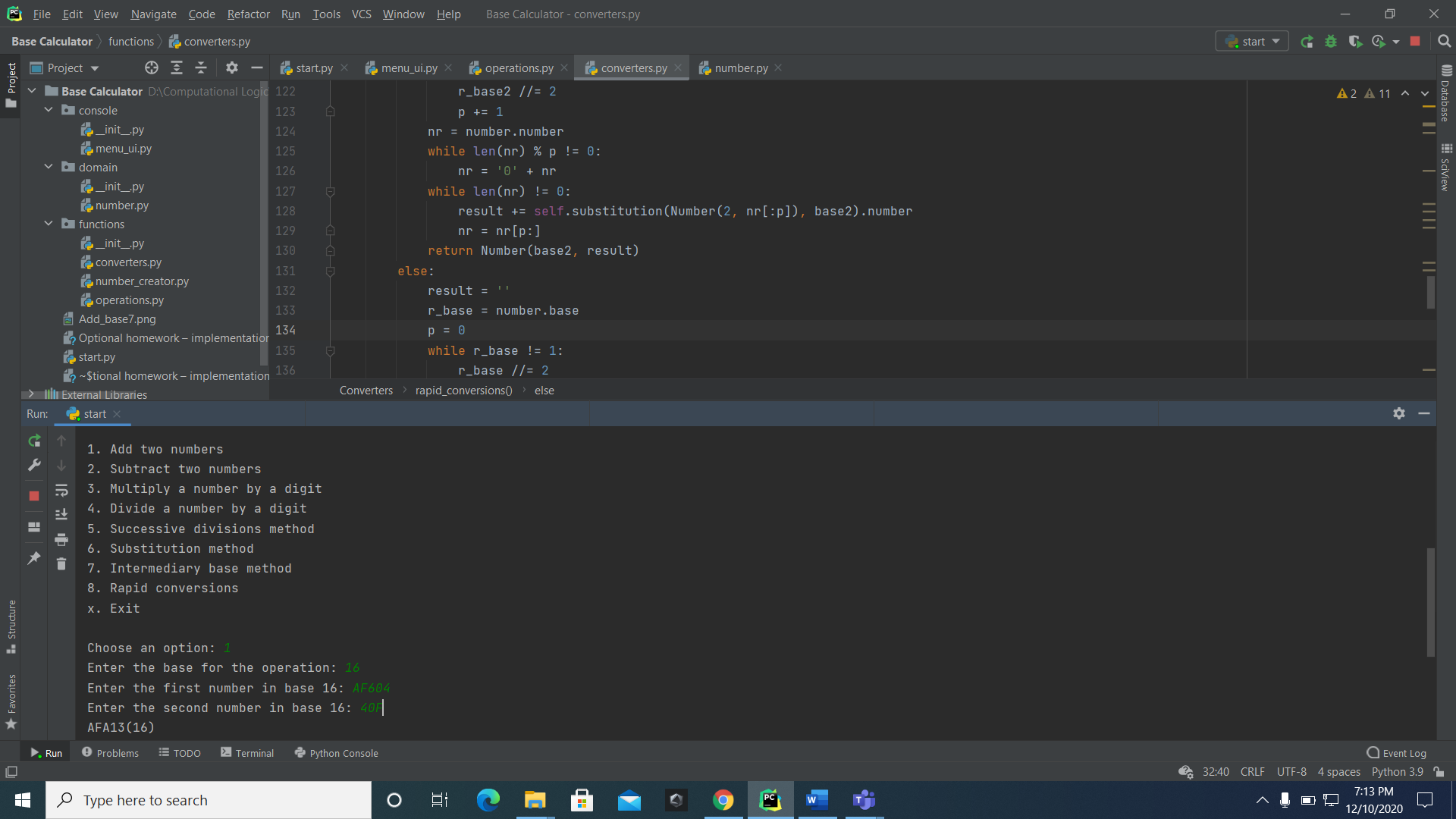
# if the source base is 2, it calculates which power of 2 is the destination base; it adds the number of 0-a necessary so the number’s length is divisible by the power, then it takes numbers in groups of the length of the power and converts them into the destination base, adding them to the result string  
 if number.base == 2:  
 result = ''  
 r\_base2 = base2  
 p = 0  
 while r\_base2 != 1:  
 r\_base2 //= 2  
 p += 1  
 nr = number.number  
 while len(nr) % p != 0:  
 nr = '0' + nr  
 while len(nr) != 0:  
 result += self.substitution(Number(2, nr[:p]), base2).number  
 nr = nr[p:]  
 return Number(base2, result)  
 else:

# if the source base is 4, 8 or 16 then it checks at which power of 2 the source base is and starts converting each digit into base 2, adding the amount of 0-s necessary such that the digit is converted into a number of length of the respective power, and each converted digit is added to the result  
 result = ''  
 r\_base = number.base  
 p = 0  
 while r\_base != 1:  
 r\_base //= 2  
 p += 1  
 for index in range(len(number)):  
 digit = self.substitution(Number(base2, number.number[index]), 2).number  
 if index != 0:  
 while len(digit) % p != 0:  
 digit = '0' + digit  
 result += digit  
 return Number(2, result)

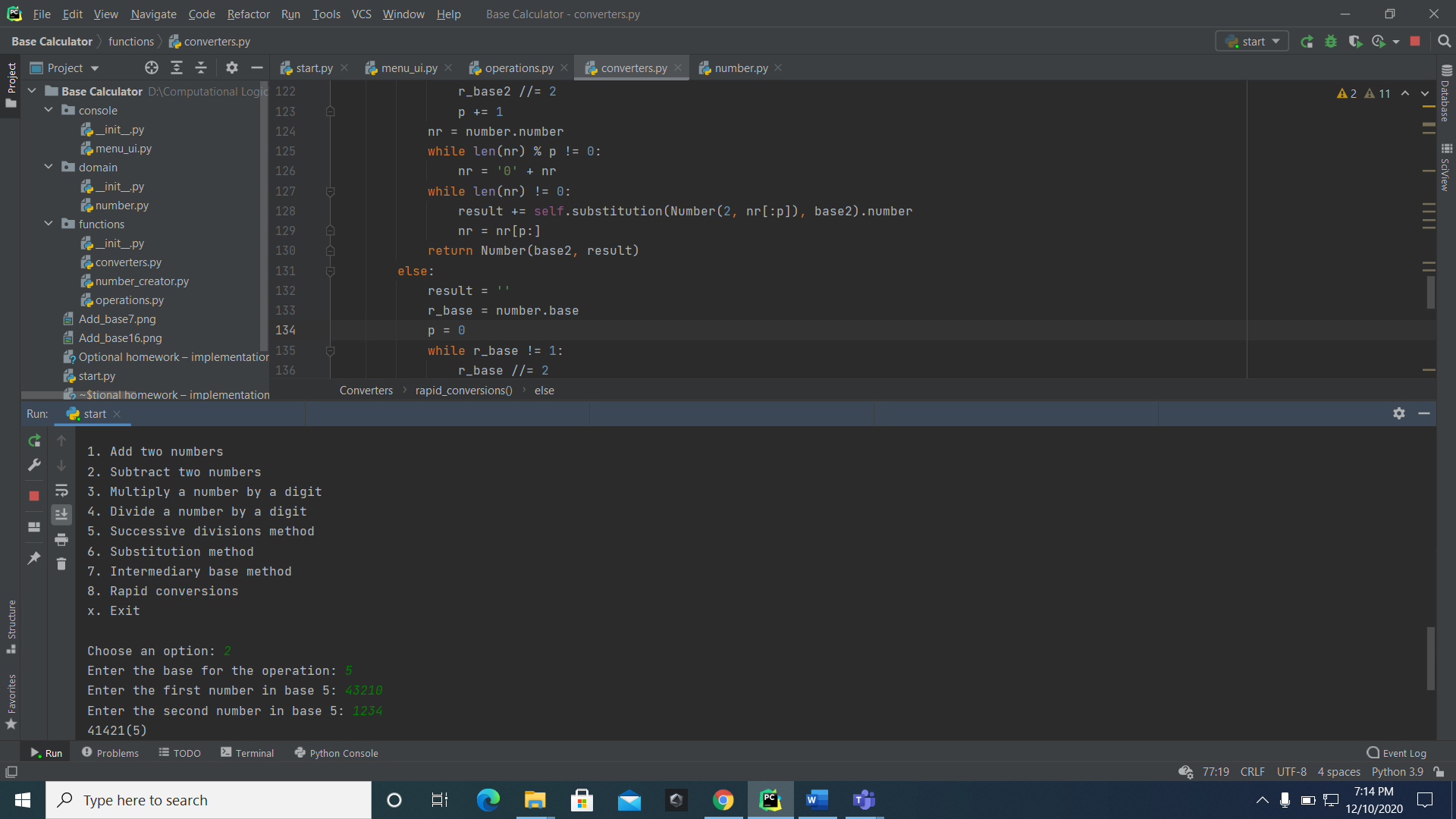
# Test data:

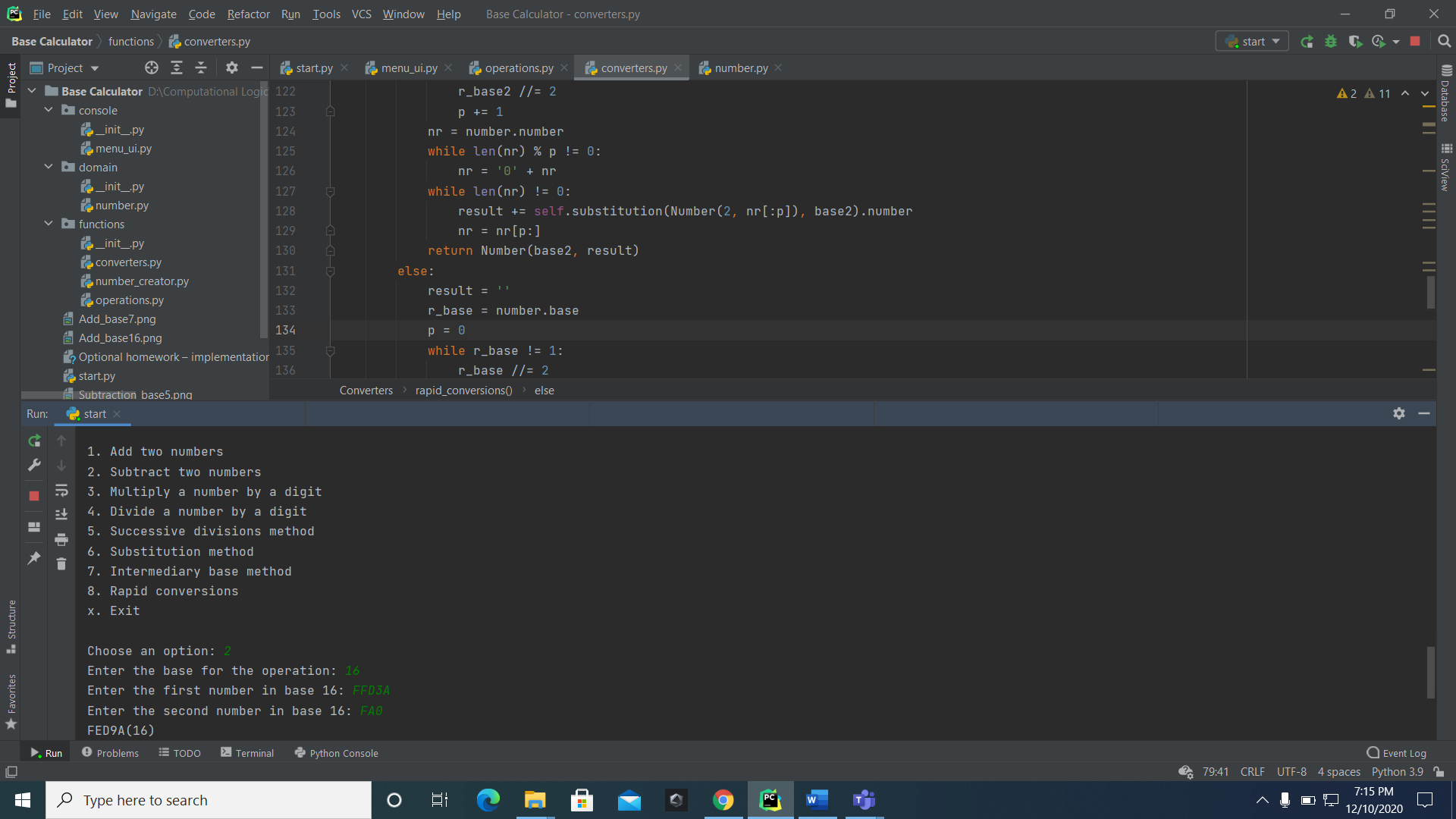
1. Addition:



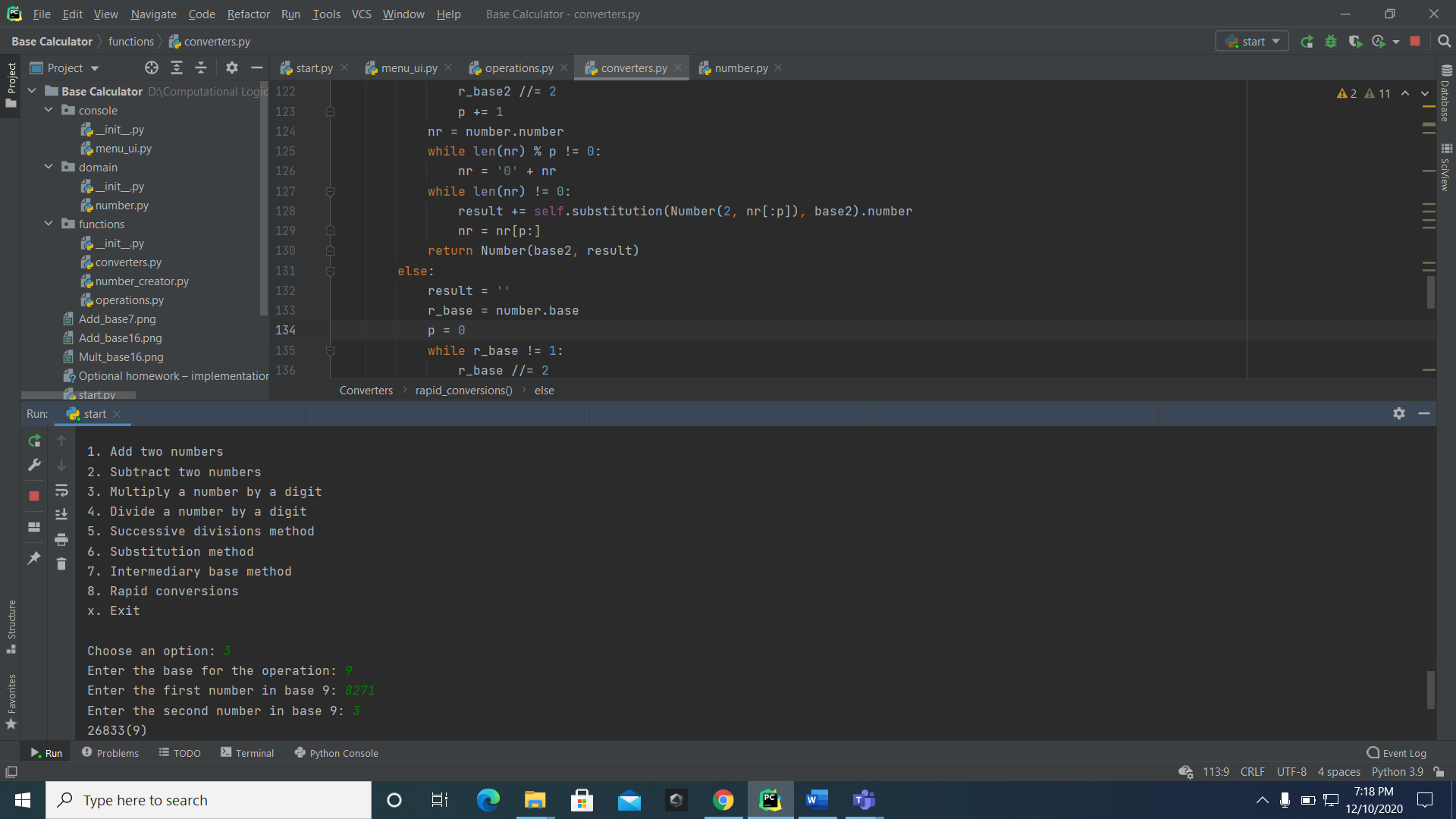


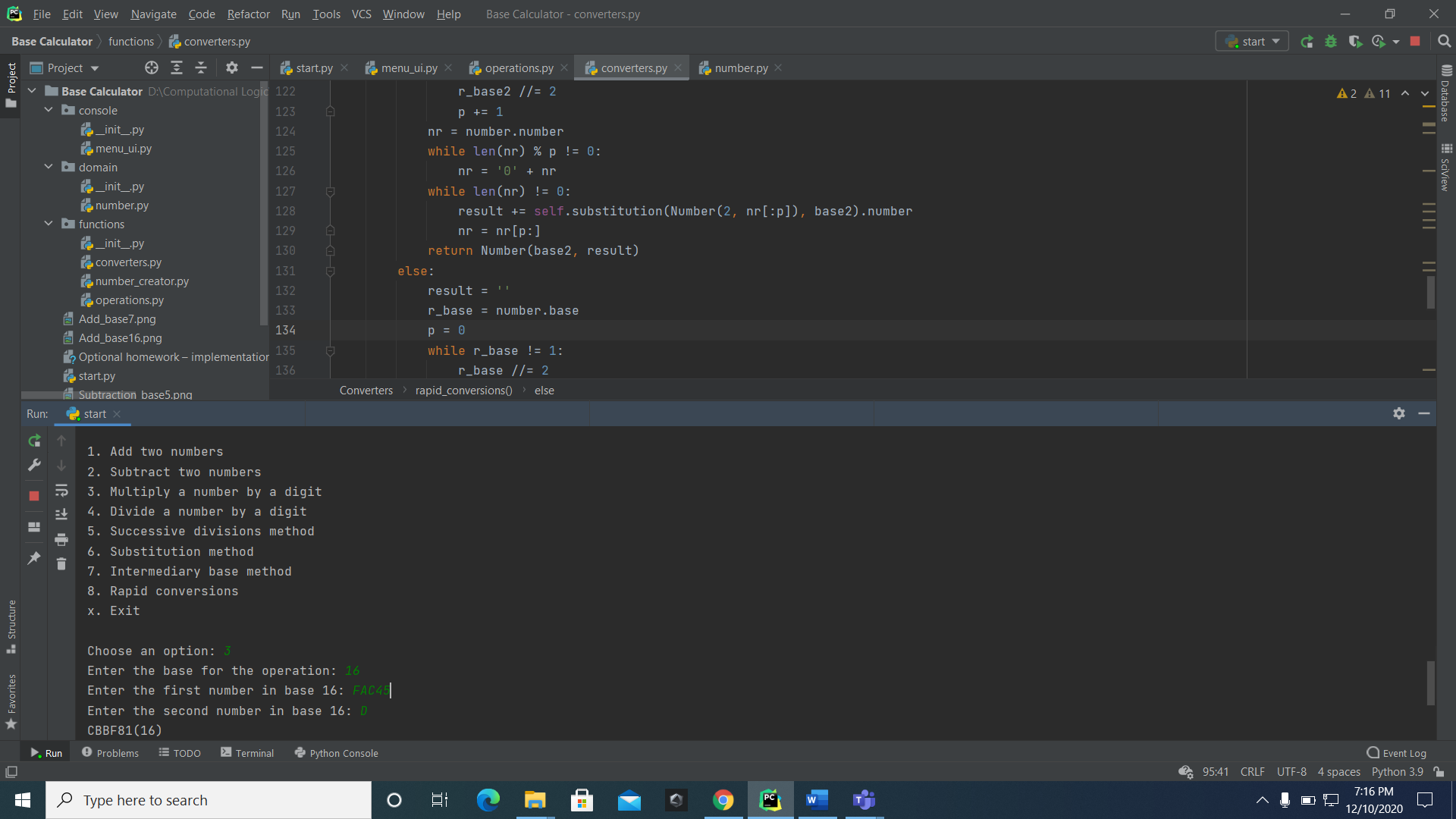
1. Subtraction:



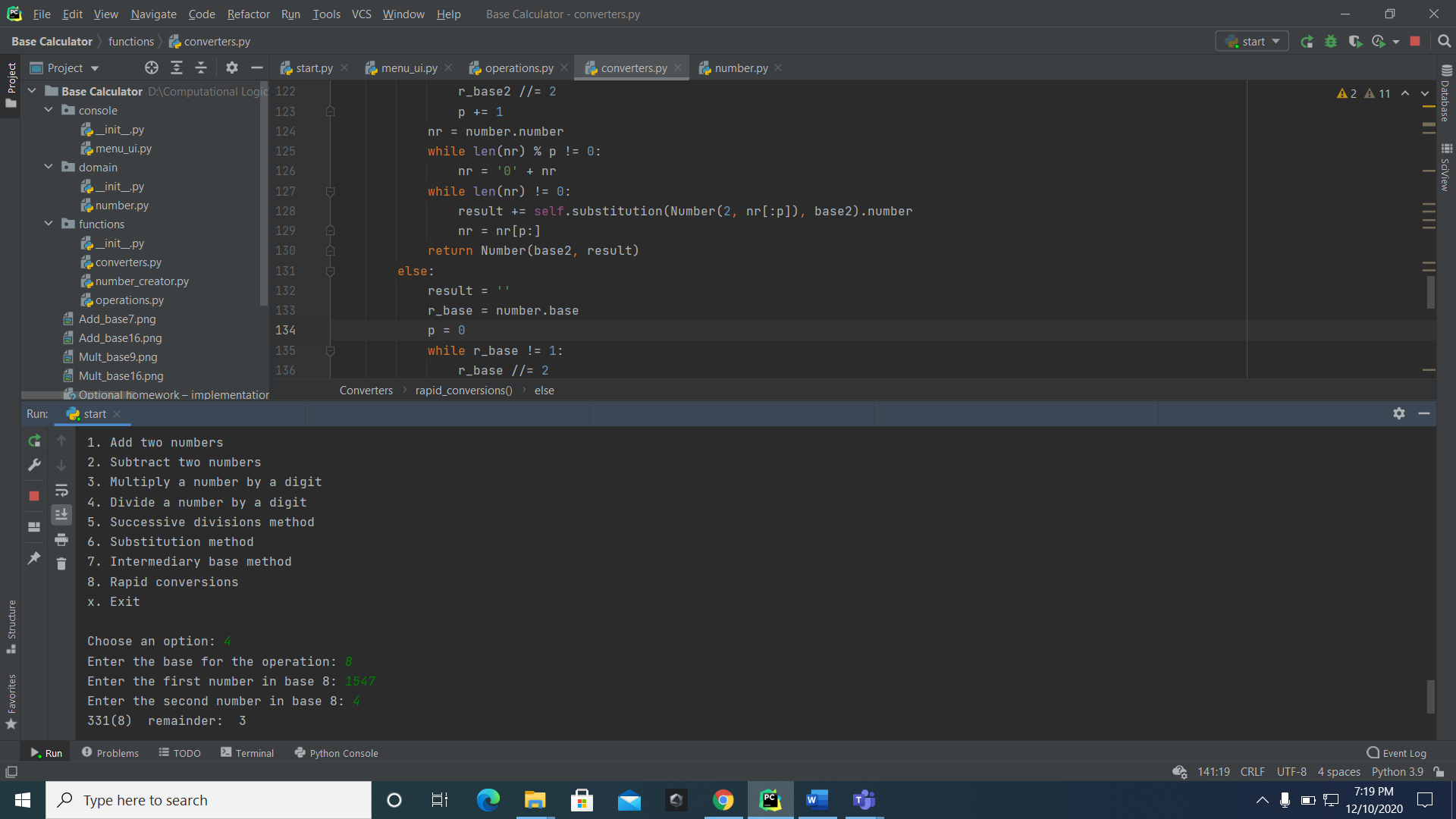


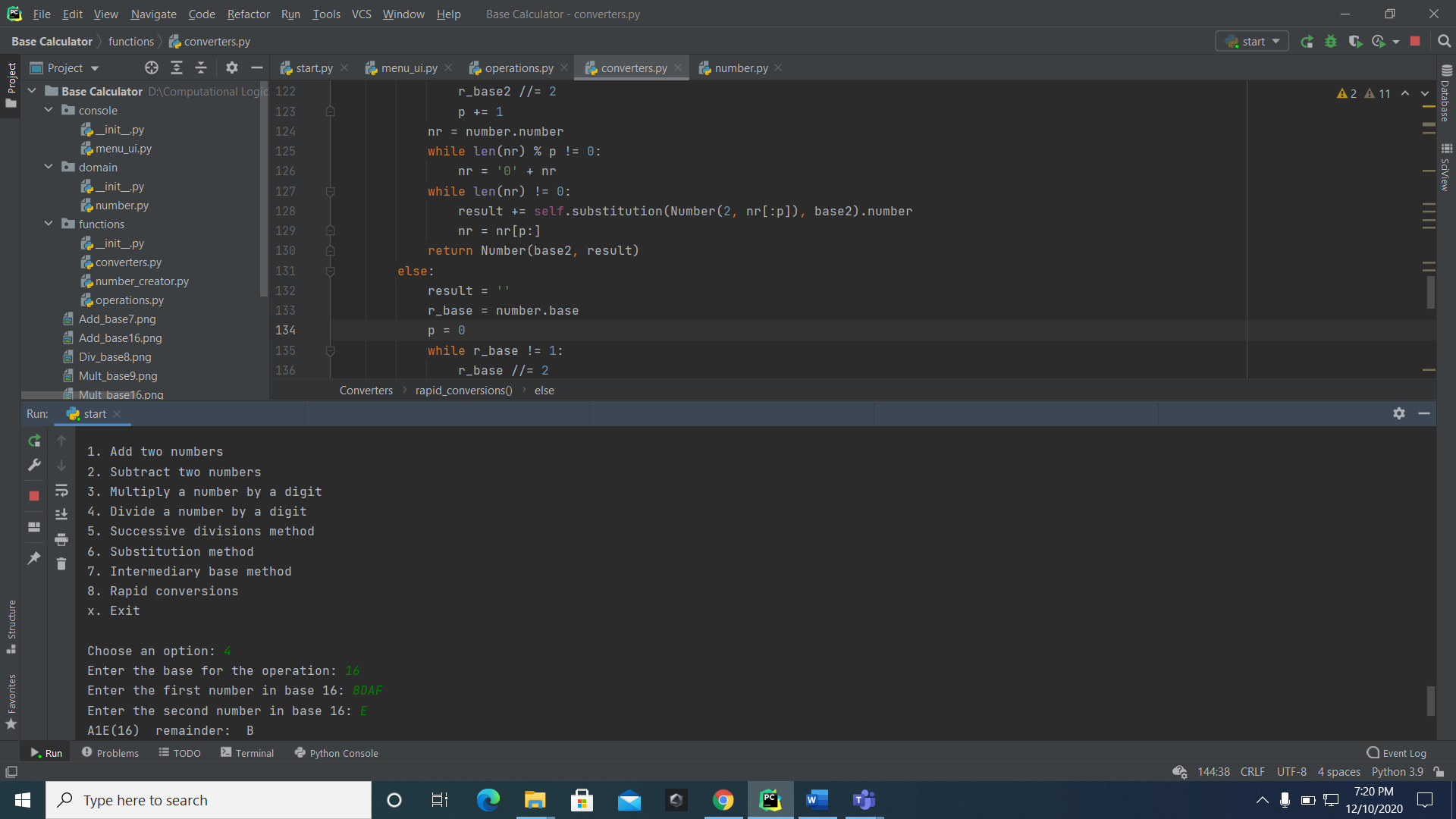
1. Multiplication:



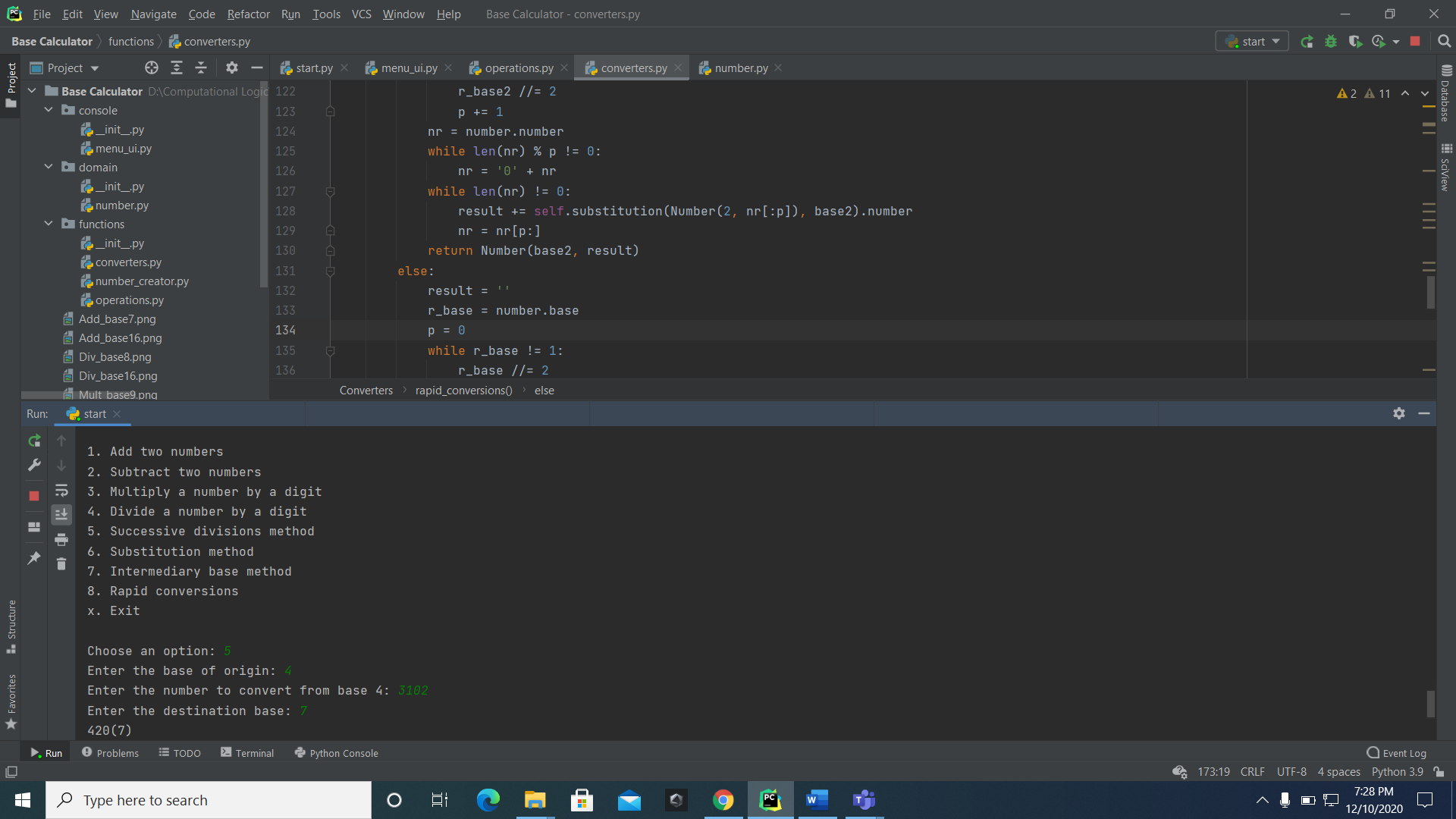


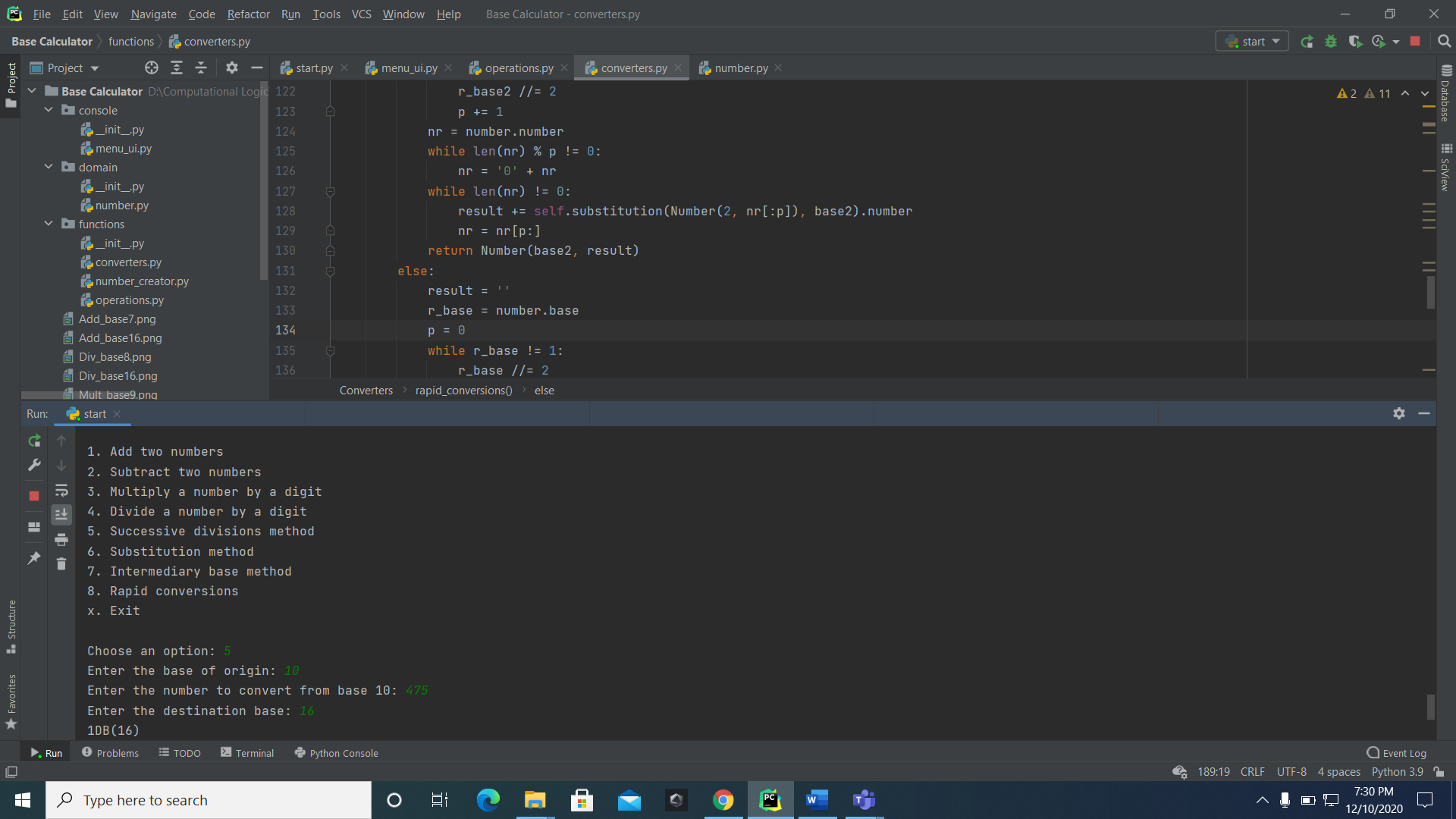
1. Division:



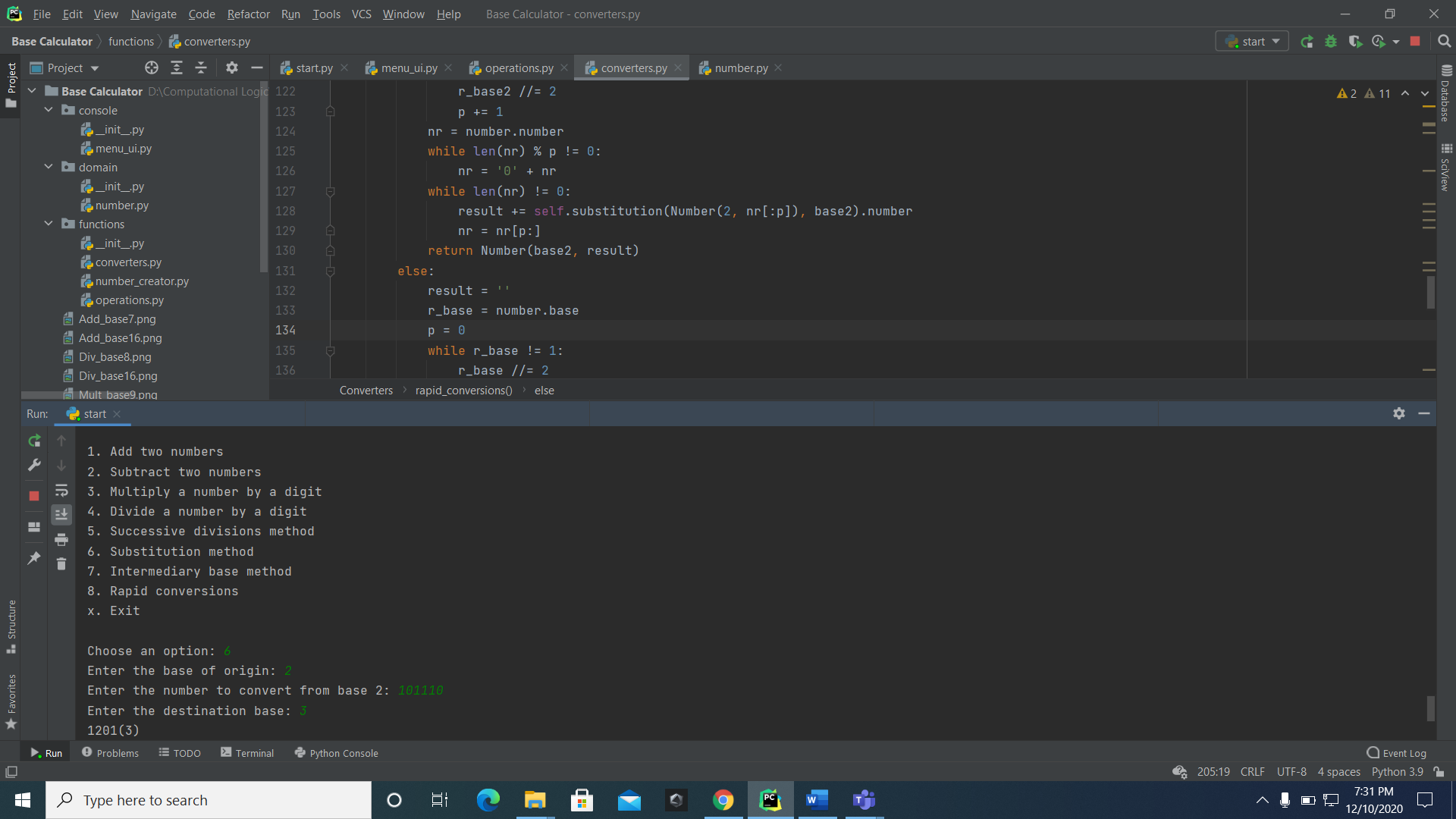


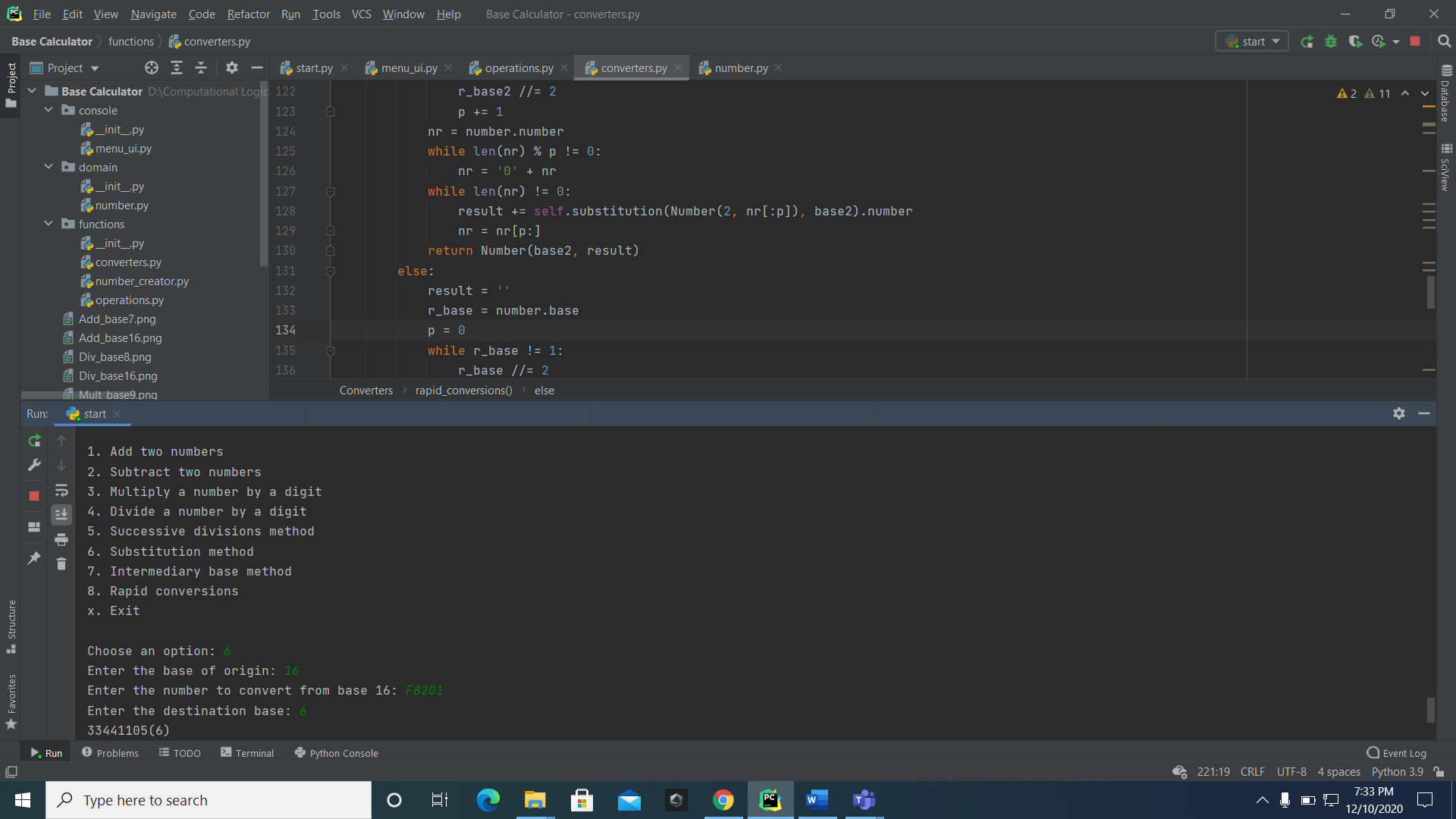
1. Successive divisions:



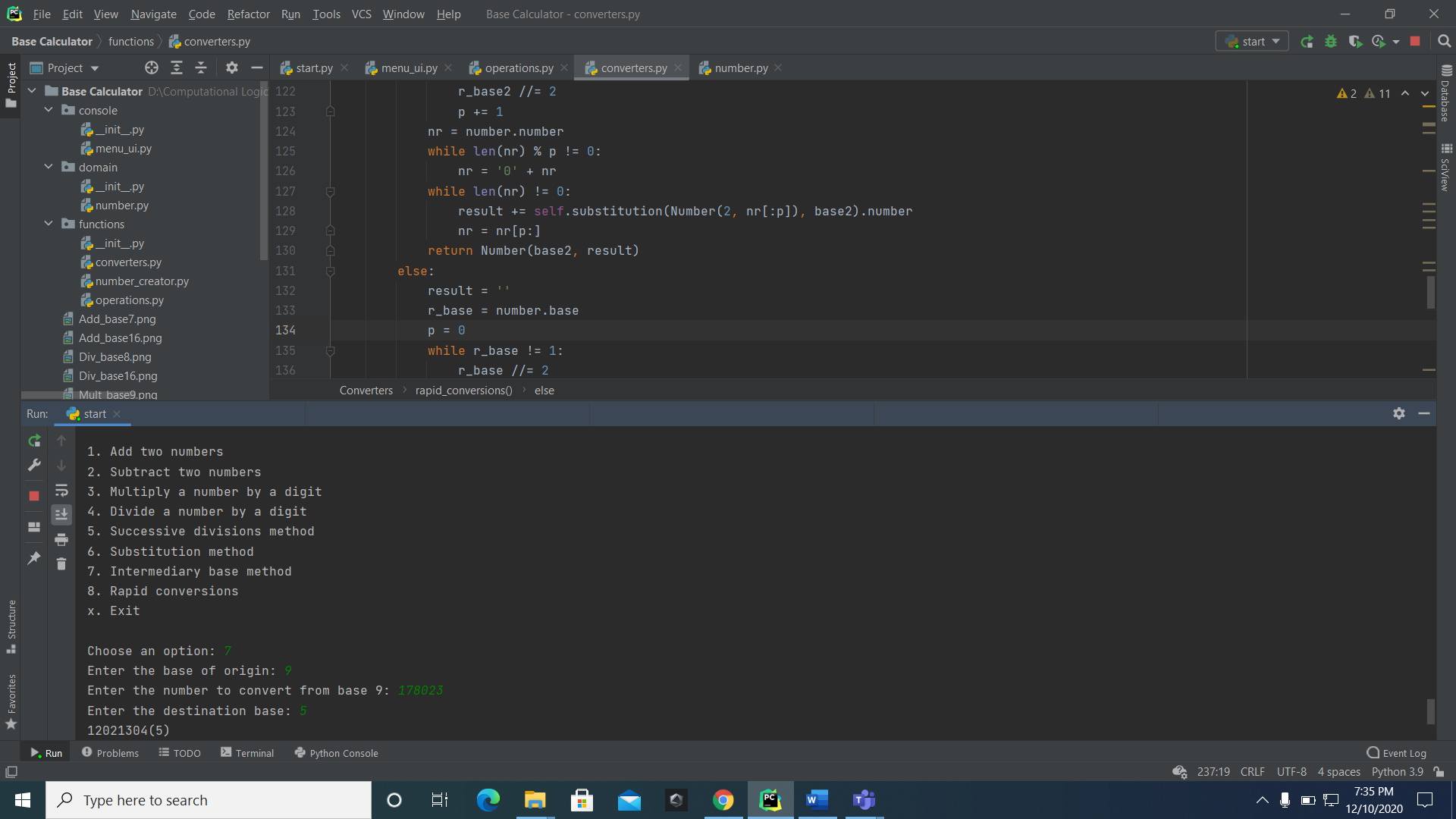


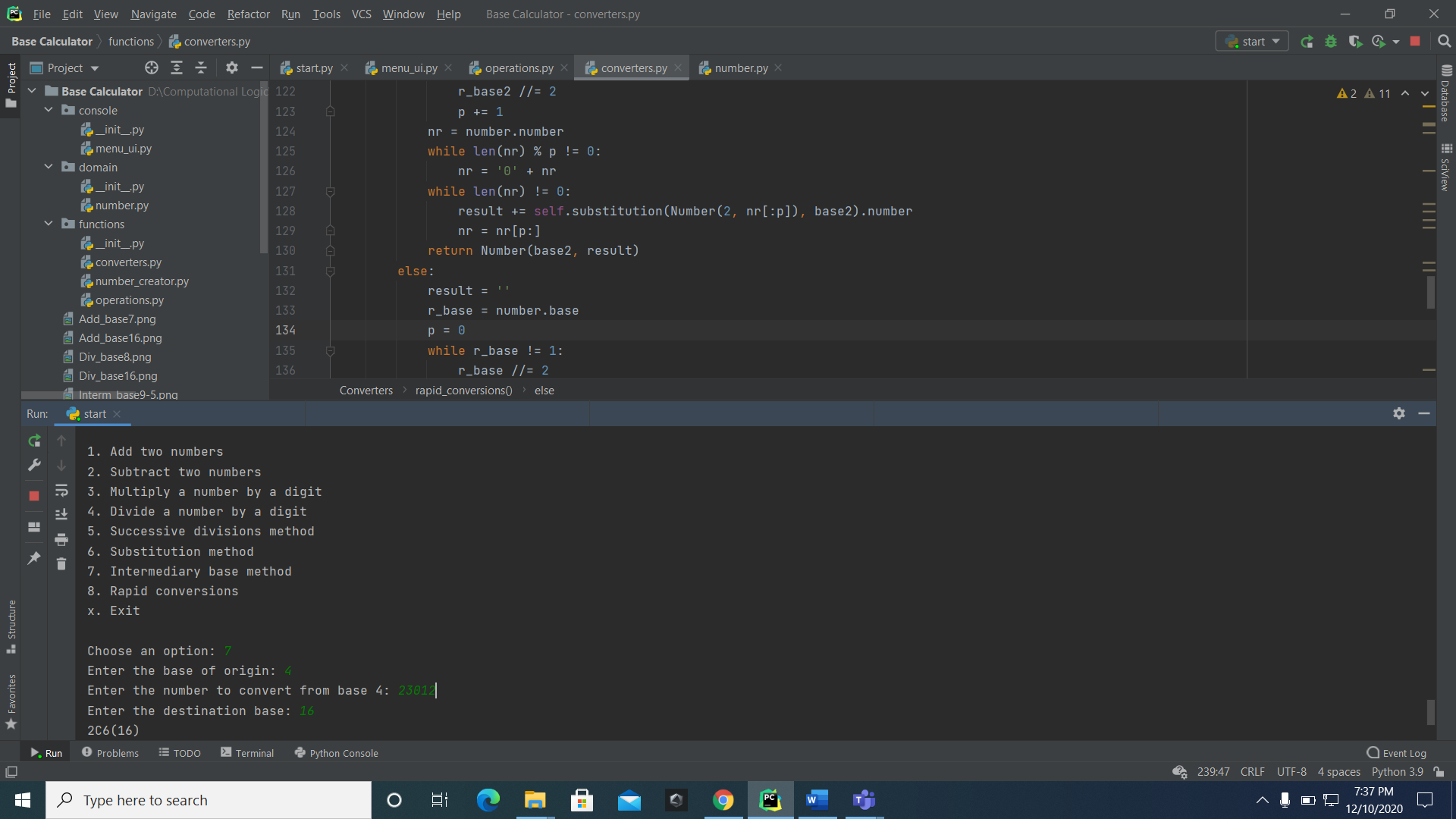
1. Substitution:





1. Intermediary base:





1. Rapid conversions:

