НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ»

ФАКУЛЬТЕТ ІНФОРМАТИКИ І ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

КАФЕДРА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

**Лабораторна робота №4**

з дисципліни **«**Системне програмування-2**»**

Виконав: студент 3 курсу

гр. ІО-81

Рибніков Євгеній

Варіант 20

Перевірив:

Павлов В. Г.

Київ 2020 р.

**Тема:** Вкладені конструкції. Оператори розгалуження. Тернарний оператор.

**Мета:** застосування у компіляторі обробки вкладених конструкцій, conditional statements (таких як if) та conditional expressions (тернарних операторів типу a ? b : c).

**Завдання:**



**Лістинг коду:**

**#dict.py**

LPAR = '('

RPAR = ')'

COLON = ':'

RETURN = 'return'

DEF = 'def'

NUM = 'num'

STRING = 'string'

ID = 'id'

INDENT = 'indent'

DQM = '"'

SQM = "'"

DEDENT = 'dedent'

NL = 'new line'

ENDMARK = 'endmark'

PLUS = '+'

EQUAL = '='

MINUS = '-'

LOGAND = 'and'

IF = 'if'

ELSE = 'else'

WHILE = 'while'

MULT = '\*'

LESS = '<'

MORE = '>'

UNAR\_NOT = 'not'

SEMICOLON = ';'

INV = '~'

**#Token**

class Token():

def \_\_init\_\_(self, tag, value=None):

self.tag = tag

self.value = value

def \_\_repr\_\_(self):

return f"{self.tag}: {self.value}"

**#Lexer.py**

from Token import Token

from dict import \*

def is\_hex(a):

set = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f'}

if a in set:

return True

else:

return False

class Lexer():

def \_\_init\_\_(self):

self.line = 1

self.index = 0

self.error = 0

self.peek = ' '

self.words = {}

self.indent = 0

self.old\_indent = 0

self.spaces = 0

def hold(self, t):

self.words[t.value] = t

def scan(self, content):

if self.index == len(content):

if self.old\_indent != 0:

self.old\_indent -= 1

return Token(DEDENT)

return Token(ENDMARK)

self.peek = content[self.index]

while self.peek == '\t':

self.indent += 1

self.index += 1

self.error += 1

self.peek = content[self.index]

while self.index < len(content) - 1:

if self.peek == ' ':

self.index += 1

self.error += 1

self.spaces += 1

self.peek = content[self.index]

else:

self.indent += self.spaces // 4

self.spaces = 0

break

if self.old\_indent < self.indent:

self.old\_indent += 1

return Token(INDENT)

elif self.old\_indent > self.indent:

self.old\_indent -= 1

return Token(DEDENT)

if self.peek == '"' or self.peek == "'":

a = ''

condition = True

while condition:

a += self.peek

self.index += 1

self.error += 1

self.peek = content[self.index]

condition = (self.peek != '"' and self.peek != "'")

if self.index == len(content) - 1:

break

a += self.peek

self.index += 1

self.error += 1

self.peek = content[self.index]

return Token(STRING, a)

if self.peek == "\n":

self.index += 1

self.error = 0

self.line += 1

self.indent = 0

return Token(NL)

if self.peek.isdigit():

k = 0

cond = True

if self.peek == '0':

self.index += 1

self.error += 1

self.peek = content[self.index]

if self.peek == 'x':

s = ''

self.index += 1

self.error += 1

self.peek = content[self.index]

while cond:

s += self.peek

if self.index == len(content) - 1:

break

self.index += 1

self.error += 1

self.peek = content[self.index]

cond = is\_hex(self.peek)

k = int(s, 16)

else:

k = 0

cond = False

while cond:

k = 10 \* k + int(self.peek)

if self.index == len(content) - 1:

break

self.index += 1

self.error += 1

self.peek = content[self.index]

cond = self.peek.isdigit()

return Token(NUM, k)

if self.peek.isalpha():

s = ''

condition = True

while condition:

s += self.peek

if self.index == len(content) - 1:

break

self.index += 1

self.error += 1

self.peek = content[self.index]

if self.peek.isalpha() or self.peek.isdigit():

condition = True

else:

condition = False

w = self.words.get(s)

if w is not None:

return w

if s == "not":

return Token(UNAR\_NOT)

elif s == "while":

return Token(WHILE)

w = Token(ID, s)

self.words[s] = w

return w

if self.peek == "~":

self.index += 1

self.error += 1

self.peek = content[self.index]

return Token(INV)

d = Token(self.peek)

if self.index < len(content) - 1:

self.index += 1

self.error += 1

self.peek = ' '

return d

**#Parser.py**

import sys

import os

from Lexer import Lexer

from Token import Token

from dict import \*

def\_list = []

var\_list = []

a = open("4-20-Python-IO-81-Rybnikov.txt", 'r')

input = a.read()

lexer = Lexer()

lexer.hold(Token(LOGAND, 'and'))

lexer.hold(Token(RETURN, 'return'))

lexer.hold(Token(DEF, "def"))

lexer.hold(Token(IF, 'if'))

lexer.hold(Token(ELSE, "else"))

lexer.hold(Token(WHILE, "while"))

unar\_op\_not = False

unar\_op\_inv = False

def match(a):

global lookahead

global input

if lookahead.tag == a:

lookahead = lexer.scan(input)

print(lookahead)

else:

raise SyntaxError

def term(t):

global lookahead

global unar\_op\_not, unar\_op\_inv

if lookahead.tag == NUM:

if unar\_op\_not:

t = "\tmov ebx, " + str(lookahead.value) + "\n" + t

t += "\tmov eax, ebx\n"

match(NUM)

unar\_op\_not = False

elif unar\_op\_inv:

t = "\tmov ebx, " + str(lookahead.value) + "\n" + t

t += "\tmov eax, ebx\n"

match(NUM)

unar\_op\_inv = False

else:

t += str(lookahead.value) + "\n"

match(NUM)

elif lookahead.tag == LPAR:

match(LPAR)

t = expression(t)

match(RPAR)

elif lookahead.tag == UNAR\_NOT:

unar\_op\_not = True

match(UNAR\_NOT)

if lookahead.value in var\_list:

match(ID)

t += "\tcmp ecx, 0\n\tpushf\n\txor ecx, ecx\n\tpopf\n\tsetz cl\n"

t = term(t)

else:

t += "\tcmp ecx, 0\n\tpushf\n\txor ecx, ecx\n\tpopf\n\tsetz cl\n"

t = term(t)

elif lookahead.tag == INV:

unar\_op\_inv = True

match(INV)

if lookahead.value in var\_list:

match(ID)

t += "\tnot eax\n"

t = term(t)

else:

t += "\tnot ebx\n"

t = term(t)

while lookahead.tag == STRING:

print("Type error: Expected int, got String in " + str(lexer.line) + " line, " + str(lexer.error) +

" index")

os.system("PAUSE")

sys.exit(1)

return t

def expression(t = ""):

global lookahead

counter = 0

t = term(t)

if lookahead.tag == PLUS:

while lookahead.tag == PLUS:

match(PLUS)

if lookahead.value in var\_list:

t += "\tadd eax, ebp\n"

t = expression(t)

else:

t += "\tadd eax, "

t = expression(t)

elif lookahead.tag == MINUS:

while lookahead.tag == MINUS:

match(MINUS)

t += "\tsub eax, "

t = expression(t)

elif lookahead.tag == MULT:

while lookahead.tag == MULT:

match(MULT)

t += "\tmov ebx, "

t = expression(t)

t += "\tmul ebx\n"

elif lookahead.tag == LOGAND:

while lookahead.tag == LOGAND:

match(LOGAND)

t += "\tcmp eax, 0\n" \

"\tjne \_case2\n" \

"\tjmp \_end\n"

t += "\_case2:\n" \

"\tmov eax, "

t = expression(t)

t += "\tcmp eax, 0\n" \

"\tmov eax, 0\n" \

"\tsetne al\n" \

"\_end:\n"

t = expression(t)

elif lookahead.tag == ID:

var\_list.append(lookahead.value)

match(ID)

if lookahead.tag == MINUS:

if lookahead.tag != EQUAL:

match(MINUS)

t += "\tsub eax, "

t = expression(t)

match(EQUAL)

t += ""

t = expression(t)

t += ""

t = expression(t)

if lookahead.tag == PLUS:

if lookahead.tag != EQUAL:

match(PLUS)

t += "\tadd eax, "

t = expression(t)

match(EQUAL)

t += ""

t = expression(t)

t += ""

t = expression(t)

if counter < 1:

if lookahead.tag == EQUAL:

counter += 1

match(EQUAL)

t += "\tpush ebp\n" \

"\tmov ebp, esp\n" \

"\tmov eax, "

t = expression(t)

t += "\tpush eax \n" \

"\tmov esp, ebp\n" \

"\tpop ebp\n"

t = expression(t)

elif lookahead.tag == IF:

match(IF)

t += "\tjmp \_post\_conditional\n"

t = expression(t)

if lookahead.tag == ELSE:

match(ELSE)

t += "\t\_e2:\n" \

"\tmov eax, "

t = expression(t)

t += "\t\_post\_conditional:\n"

elif lookahead.tag == WHILE:

while lookahead.tag == WHILE:

match(WHILE)

if lookahead.value in var\_list:

match(ID)

else:

print(f"Unknown variable in {str(lexer.line)} line, {str(lexer.error)} index")

os.system("PAUSE")

sys.exit(1)

if lookahead.tag == LESS:

match(LESS)

if lookahead.tag == MORE:

match(MORE)

t += "\tmov ecx, "

t = expression(t)

t += "\tcycle:\n"

try:

match(COLON)

except SyntaxError:

print(f"SyntaxError in {str(lexer.line - 1)} line, {str(lexer.error + 9)} index. Expect \":\".")

match(NL)

match(INDENT)

t = expression(t)

t += "\tcmp eax, ecx\n" \

"\tjne cycle\n"

elif lookahead.tag == RETURN:

match(RETURN)

if lookahead.tag != UNAR\_NOT and lookahead.tag != ID:

t += "\tmov eax, "

if lookahead.value in var\_list:

match(ID)

if lookahead.tag == ID and lookahead.value not in var\_list:

if lookahead.value not in def\_list:

print(f"Unknown variable in {str(lexer.line)} line, {str(lexer.error)} index")

os.system("PAUSE")

sys.exit(1)

if lookahead.value in def\_list:

match(ID)

match(LPAR)

match(RPAR)

if lookahead.tag == ID and lookahead.value not in def\_list:

print("Unknown function call in " + str(lexer.line) + " line, " + str(

lexer.error) + " index")

os.system("PAUSE")

sys.exit(1)

t = expression(t)

while lookahead.tag == NL:

match(NL)

while lookahead.tag == NL:

match(NL)

if lookahead.tag == DEDENT:

match(DEDENT)

return t

def start(c):

global lookahead

inside = []

outside = []

outside\_s = ''

inside\_s = ''

counter\_of\_names = 0

counter\_in = 0

counter\_out = 0

lookahead = lexer.scan(input)

while lookahead.tag != ENDMARK:

while lookahead.tag == DEF:

counter\_out += 1

match(DEF)

invoker = ''

invoker += f"\n{lookahead.value} proc\n"

def\_list.append(lookahead.value)

match(ID)

match(LPAR)

match(RPAR)

match(COLON)

match(NL)

match(INDENT)

invoker += expression()

if def\_list[counter\_of\_names] != "main":

invoker += f"\tinvoke MessageBox,0,str$(eax), ADDR Caption, MB\_OK\n"

if def\_list[counter\_of\_names] == "main":

invoker += f"\tcall {def\_list[counter\_of\_names-1]}\n"

invoker += f"\tret\n"

while lookahead.tag == DEDENT:

match(DEDENT)

invoker += f"{def\_list[counter\_of\_names]} endp\n"

counter\_of\_names += 1

while lookahead.tag == NL:

match(NL)

outside.append(invoker)

while lookahead.tag == ID:

invoker = ''

counter\_in += 1

if lookahead.value in def\_list:

invoker += f"\tinvoke {lookahead.value}\n"

match(ID)

match(LPAR)

match(RPAR)

while lookahead.tag == NL:

match(NL)

else:

invoker = expression(invoker)

inside.append(invoker)

match(ENDMARK)

for i in range(counter\_out):

outside\_s += outside[i]

for j in range(counter\_in):

inside\_s += inside[j]

c = c + outside\_s + "\nstart:\n" + inside\_s + "\tinvoke ExitProcess, 0\n

**#main.py**

import os

from Parser import start

a = f".386\n" \

f".model flat, stdcall\n\n" \

f"option casemap: none;\n" \

f"include E:\masm32\include\kernel32.inc\n" \

f"include E:\masm32\include\windows.inc\n" \

f"include E:\masm32\include\masm32rt.inc\n" \

f"include E:\masm32\include\\user32.inc\n" \

f"includelib E:\masm32\lib\kernel32.lib\n" \

f"includelib E:\masm32\lib\\user32.lib\n" \

f".data\n" \

f"Caption db 'Rybnikov', 0\n" \

f".code\n\n"

a = start(a)

b = open("4-20-Python-IO-81-Rybnikov.asm", "w")

b.write(a)

b.close()

c = open("4-20-Python-IO-81-Rybnikov.asm", "r")

line = c.readline()

while line:

print(line)

line = c.readline()

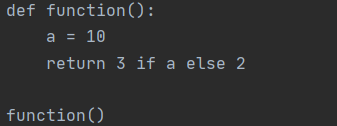
c.close()

os.system("PAUSE")

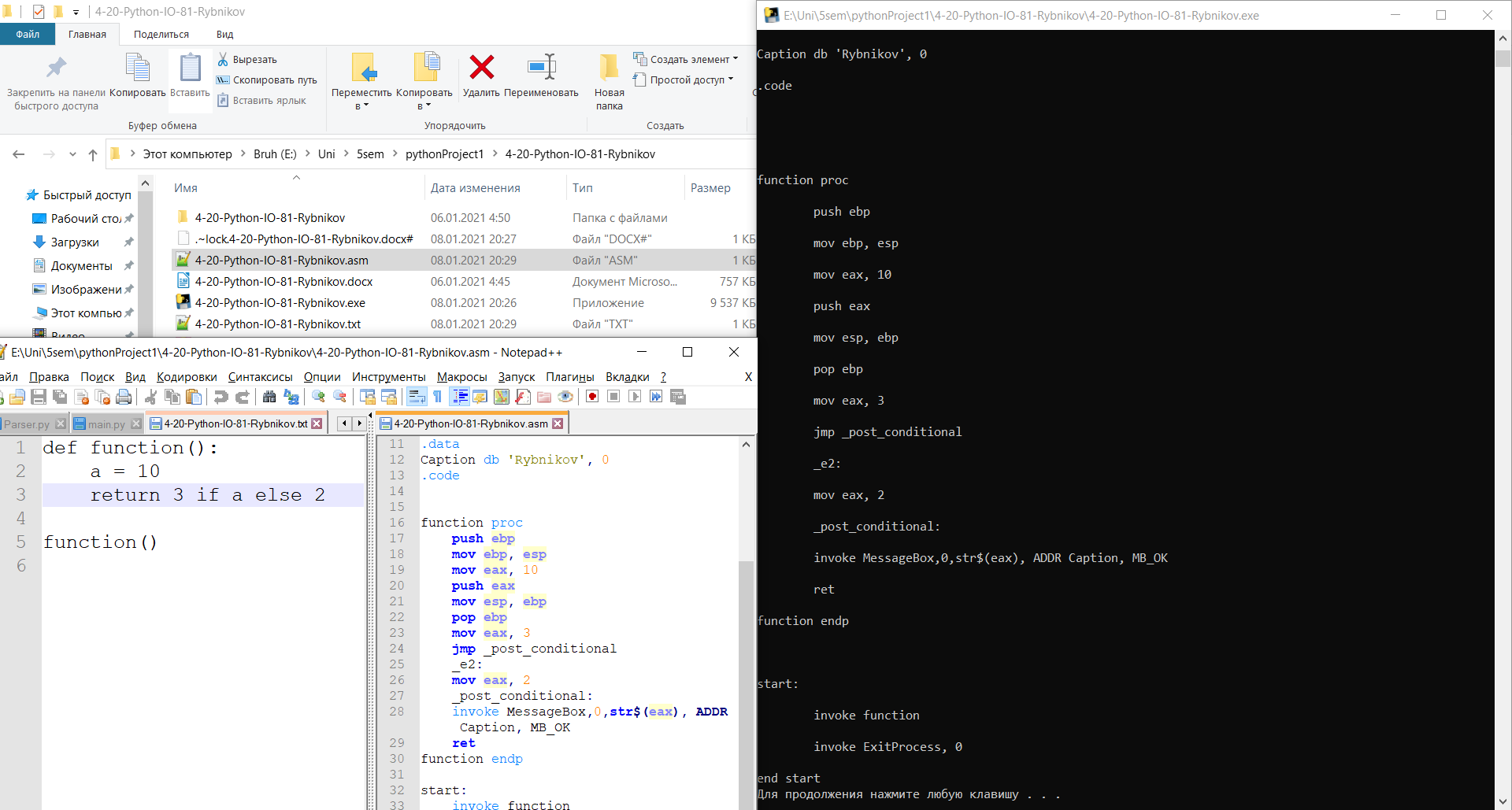
**Контрольні приклади**

**Приклад 1**

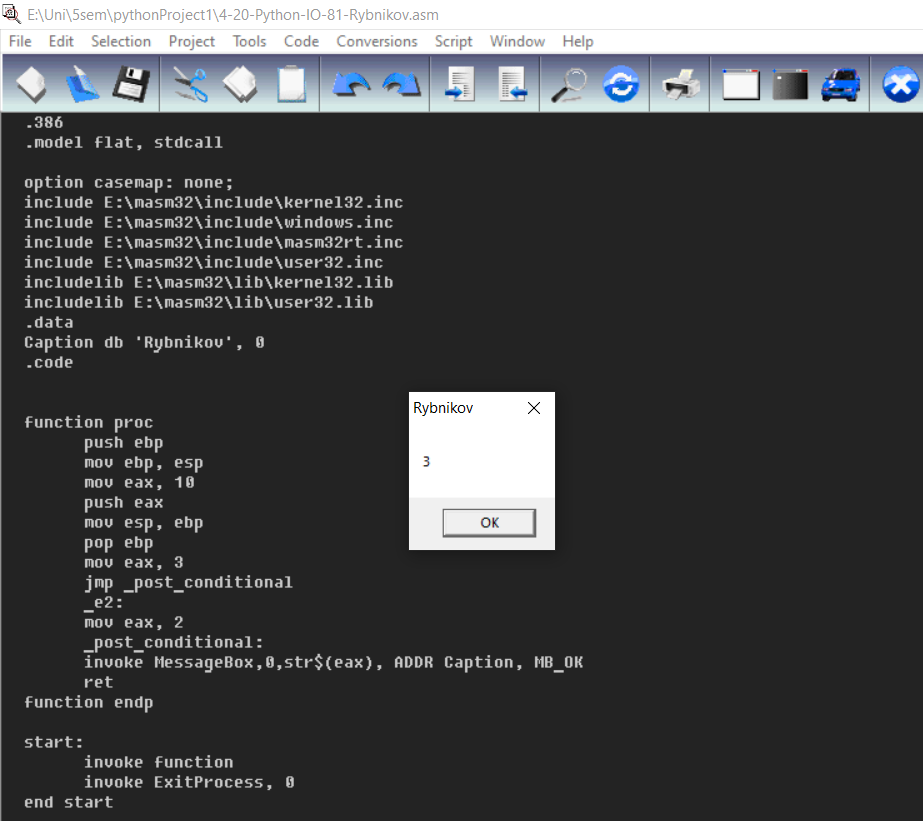
Вхідний код:



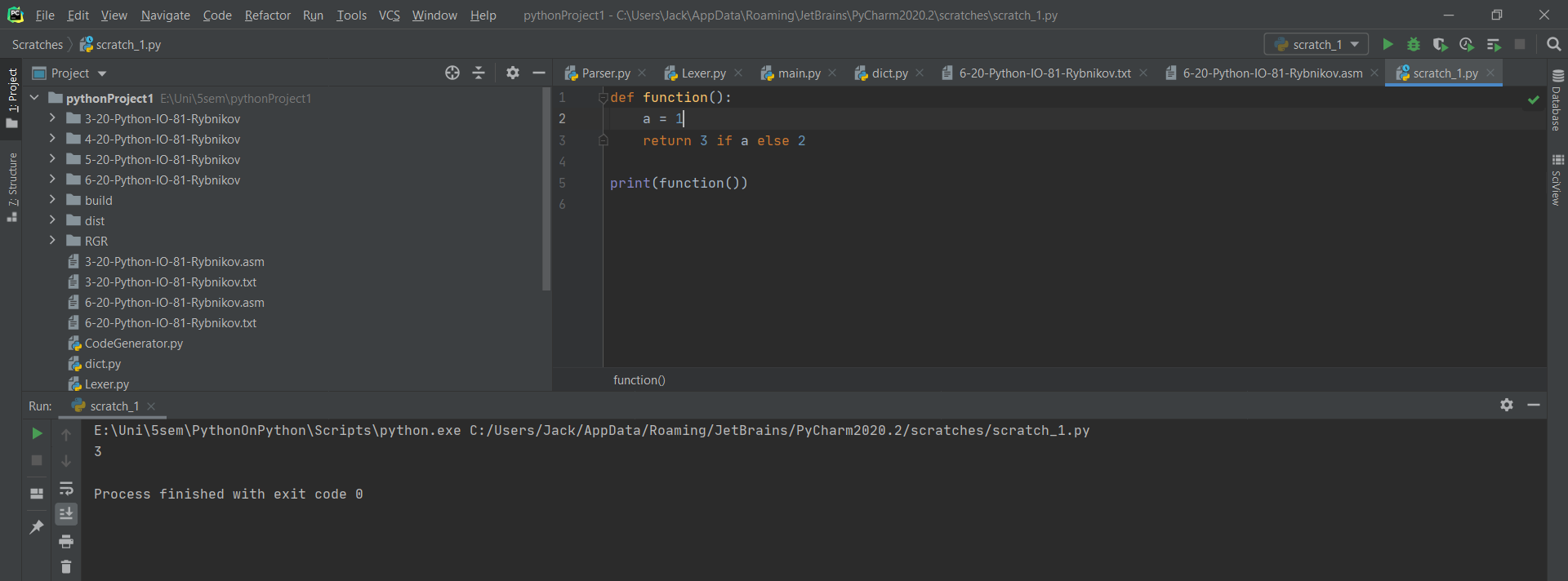
Вихідний код:



Перевірка роботи вихідного коду:

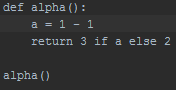


Перевірка правильності розрахунків на мові Python:

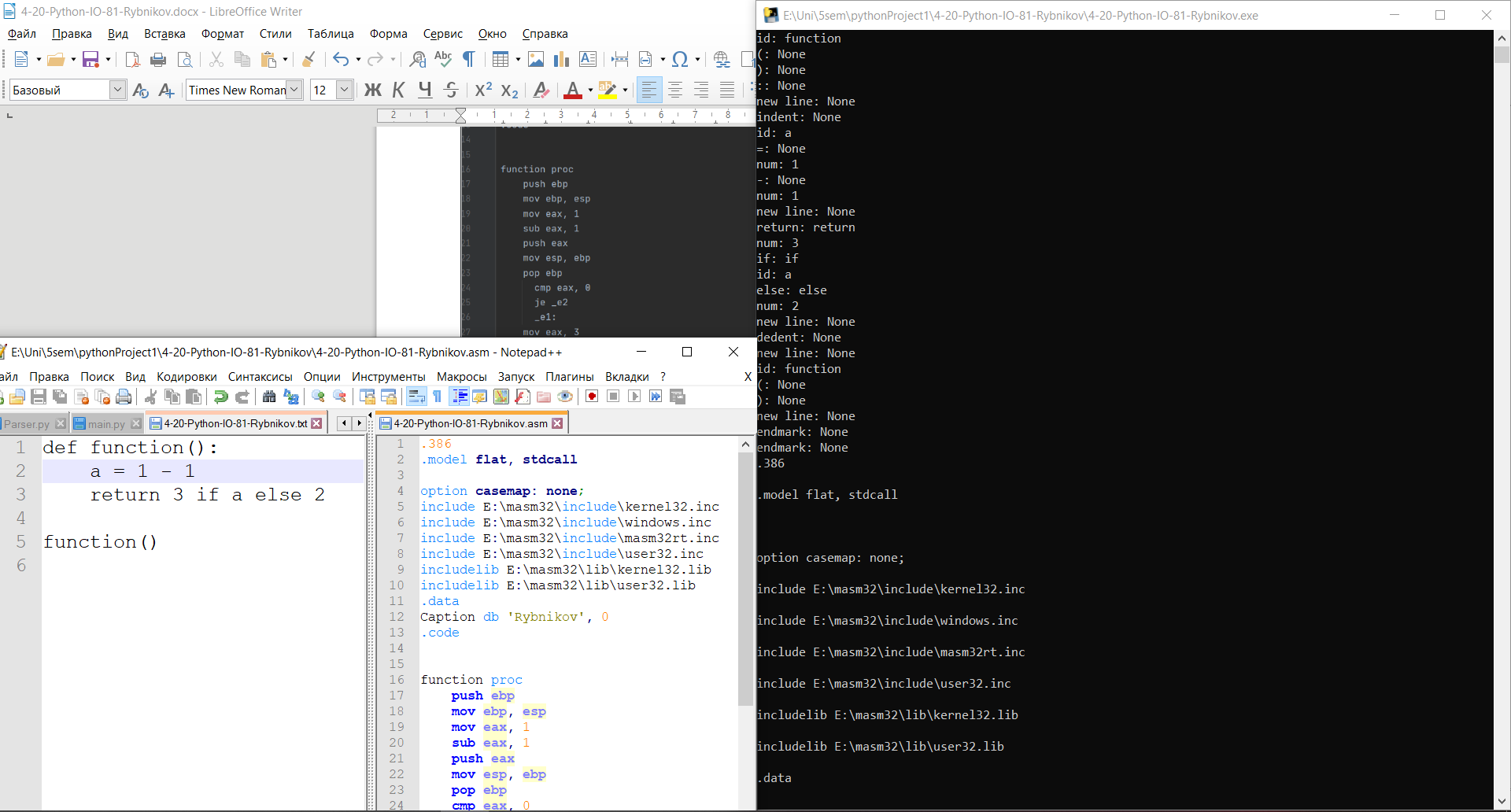


**Приклад 2**

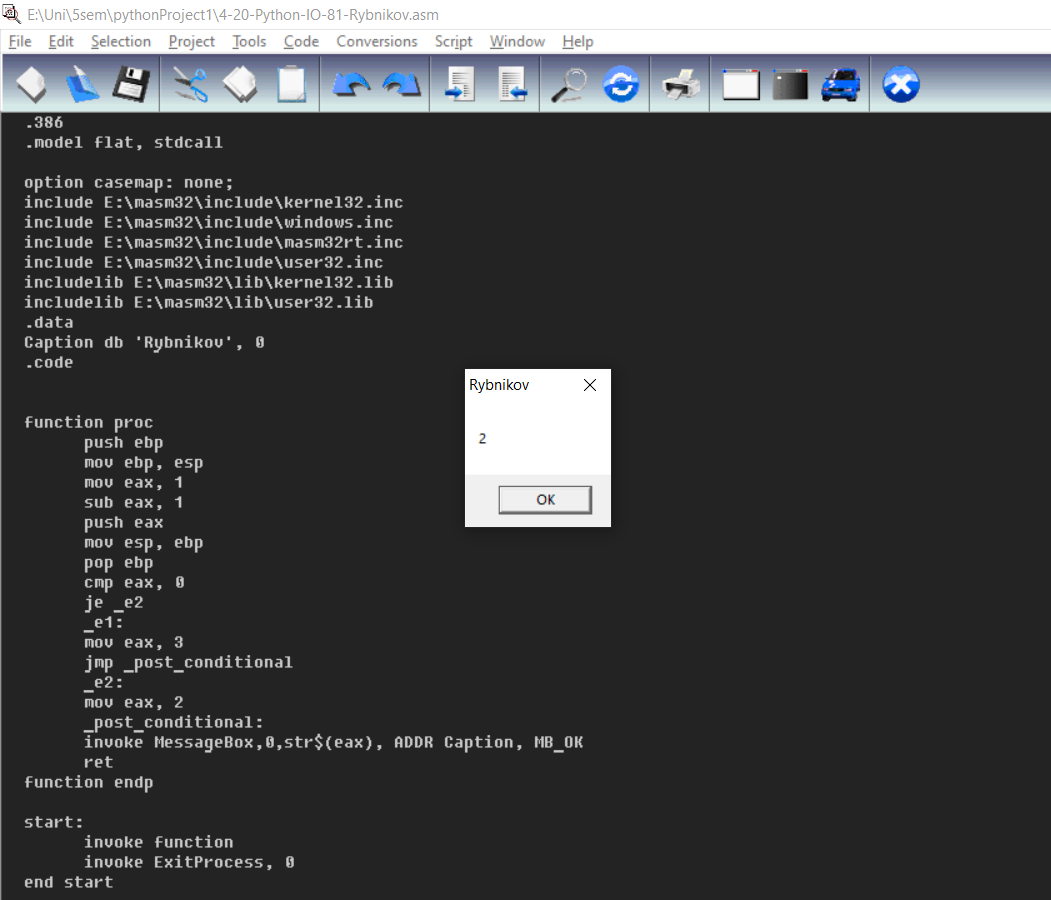
Вхідний код:



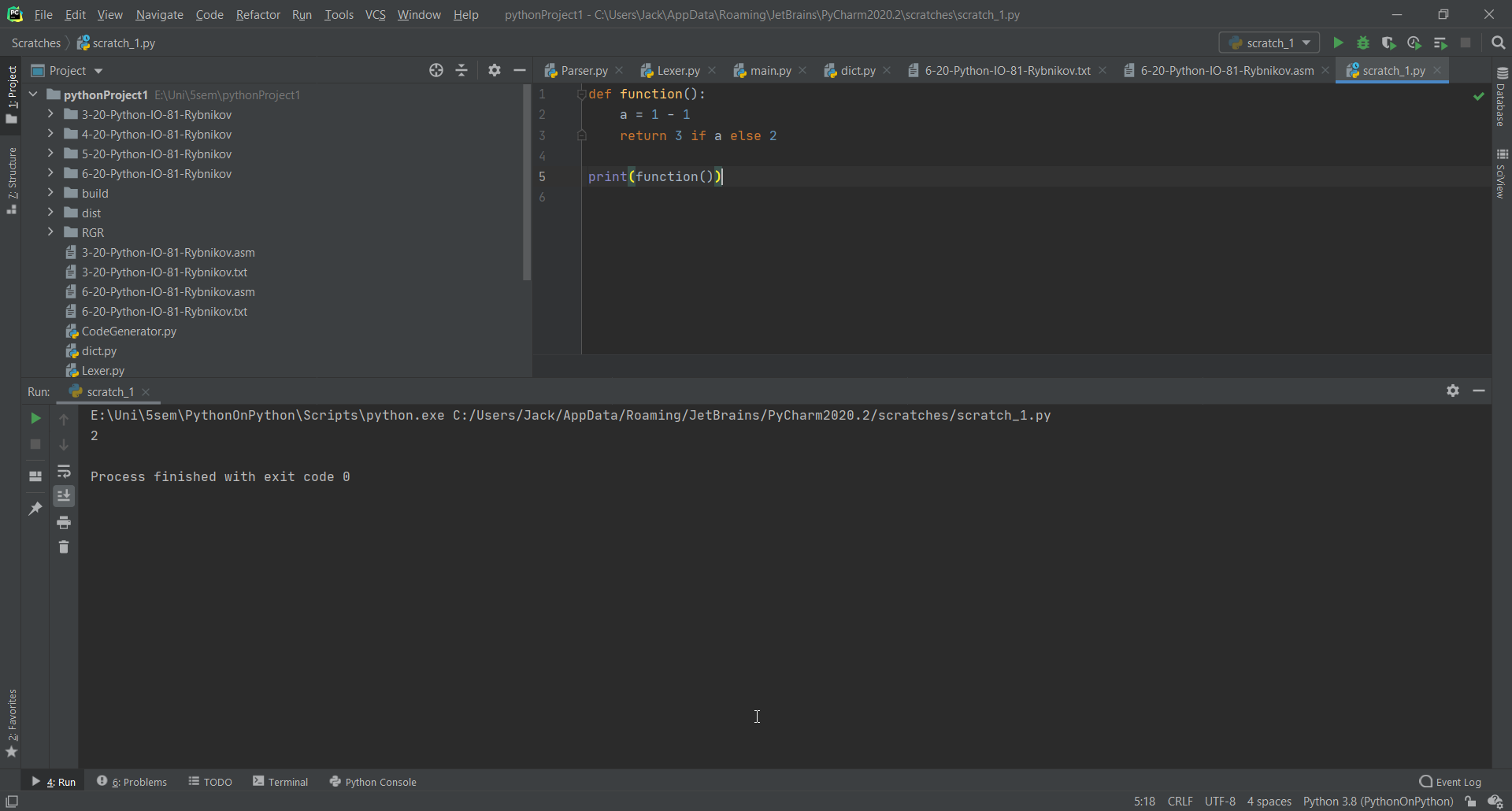
Вихідний код:



Перевірка роботи вихідного коду:

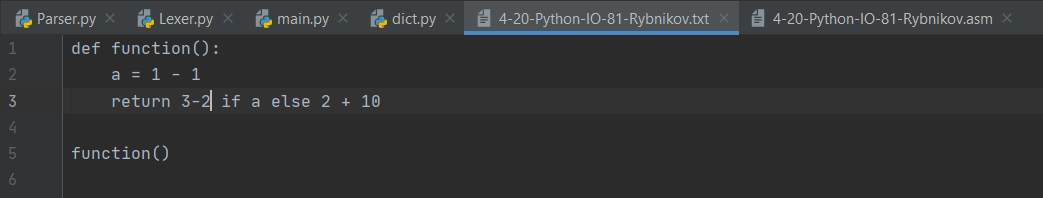


Перевірка правильності розрахунків на мові Python:

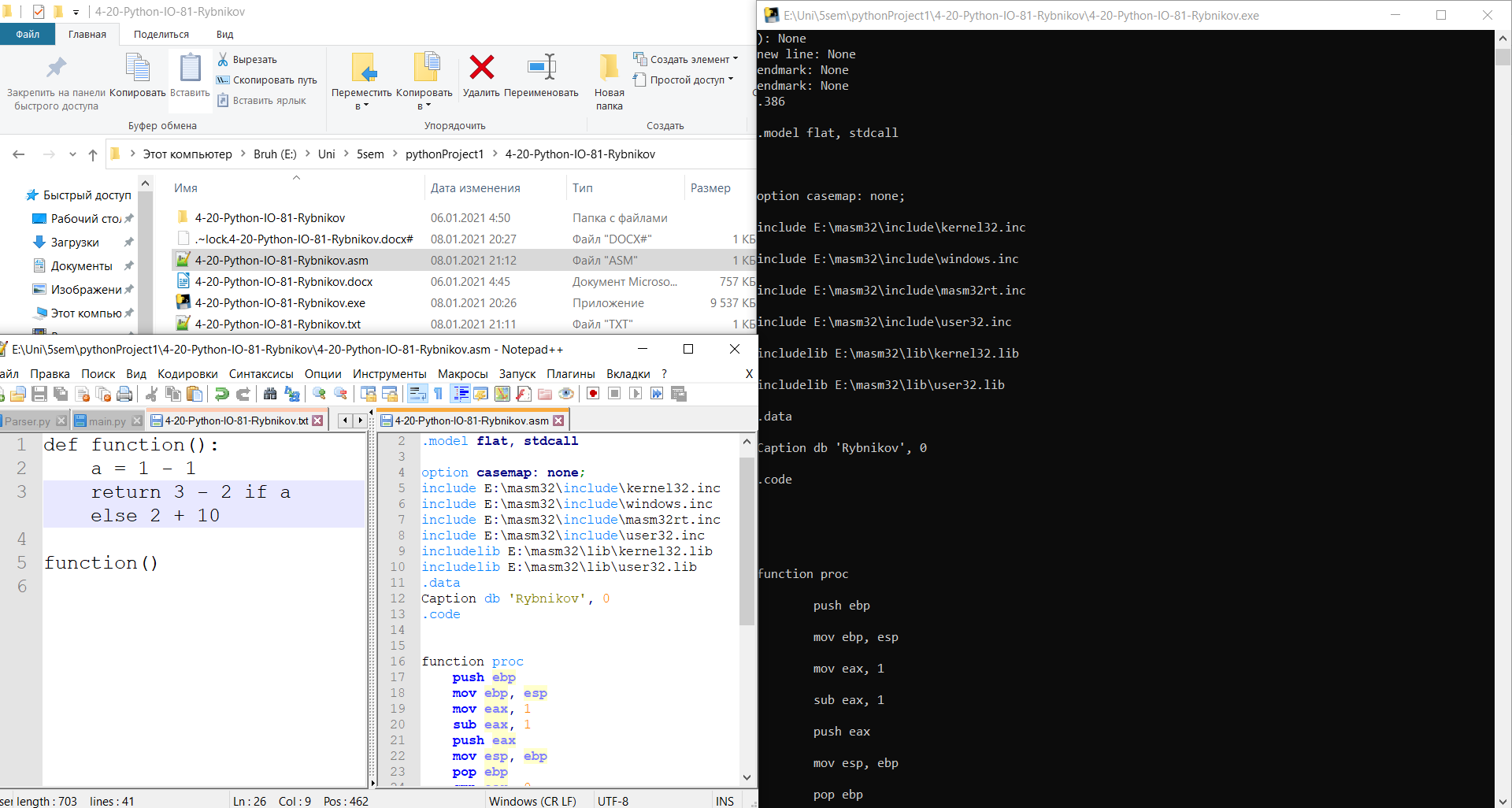


**Приклад 3**

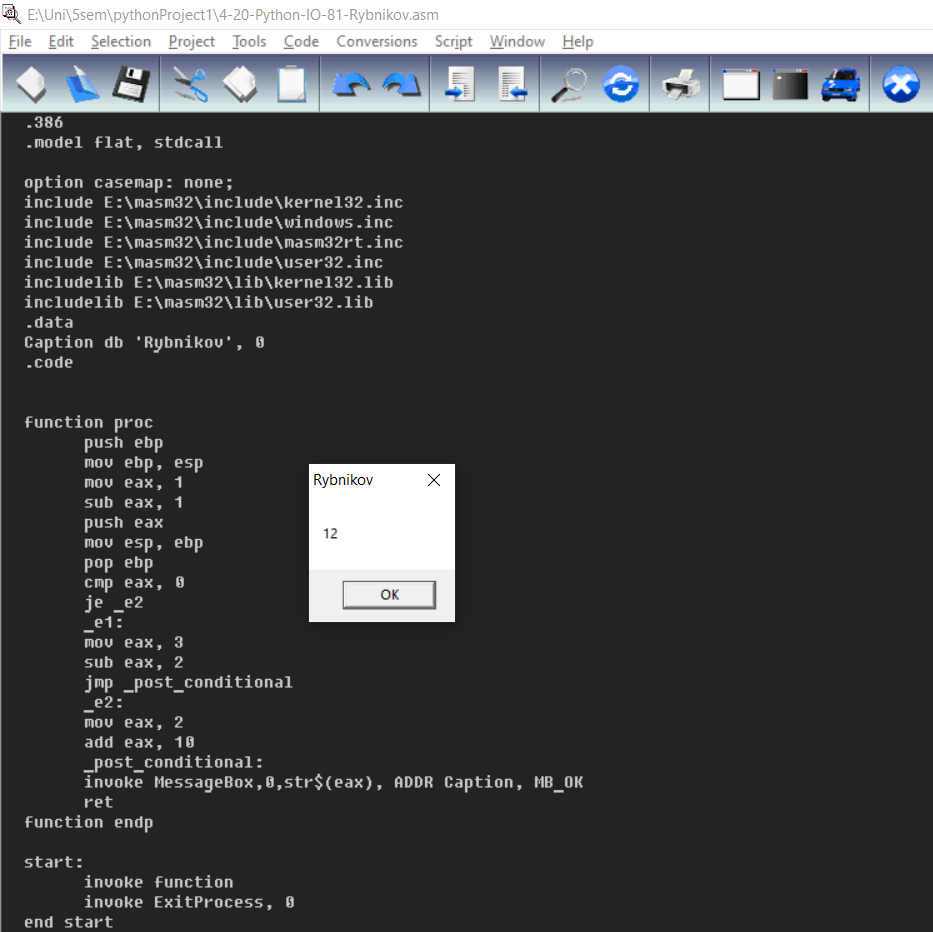
Вхідний код:



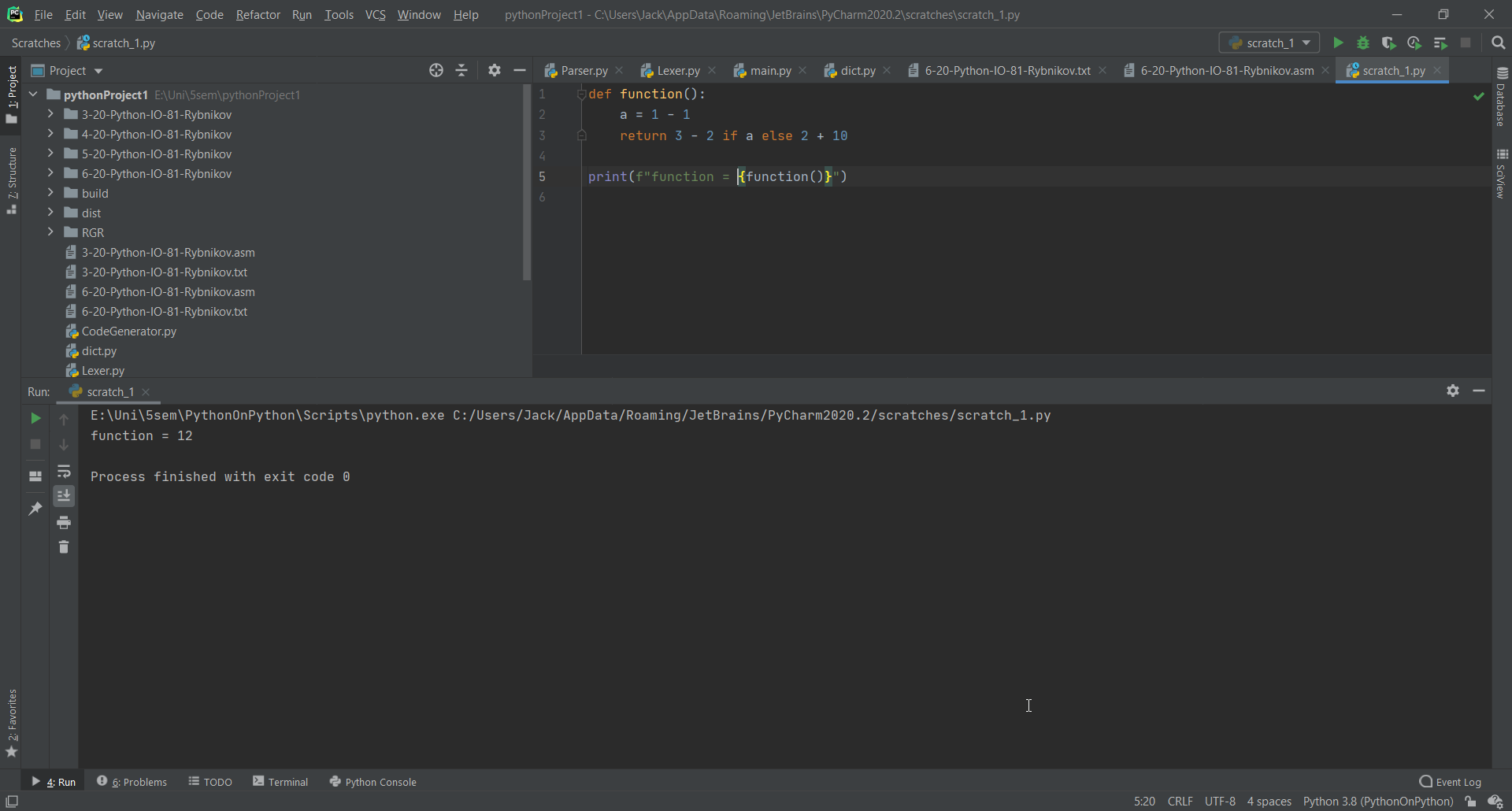
Вихідний код:



Перевірка роботи вихідного коду:

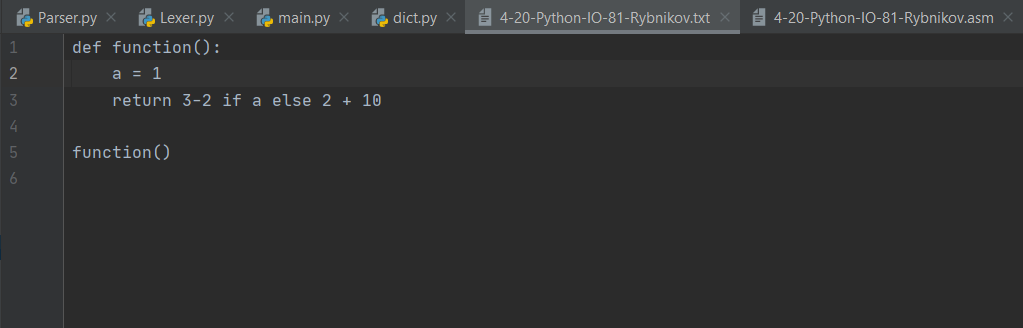


Перевірка правильності розрахунків на мові Python:

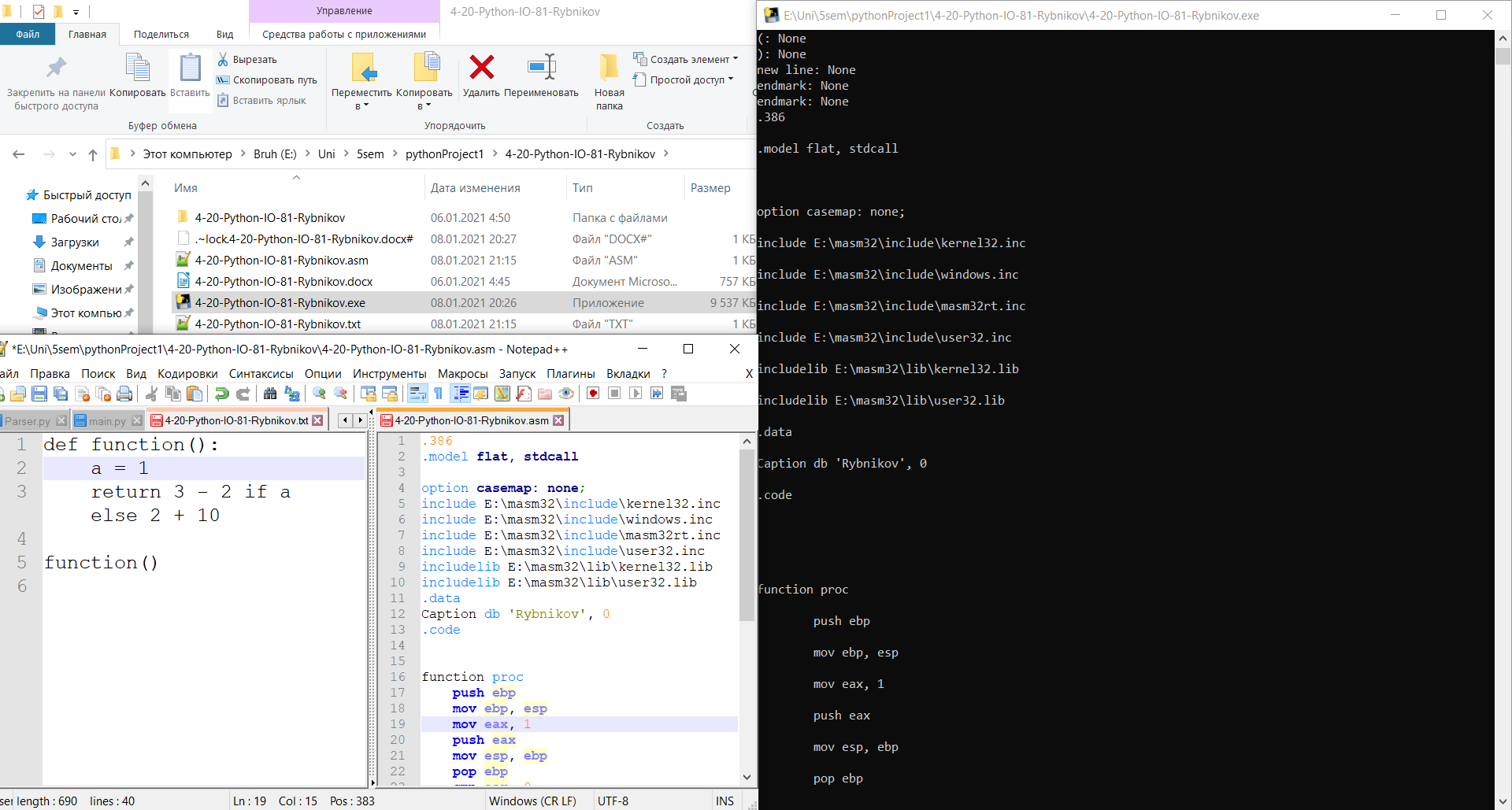


**Приклад 4**

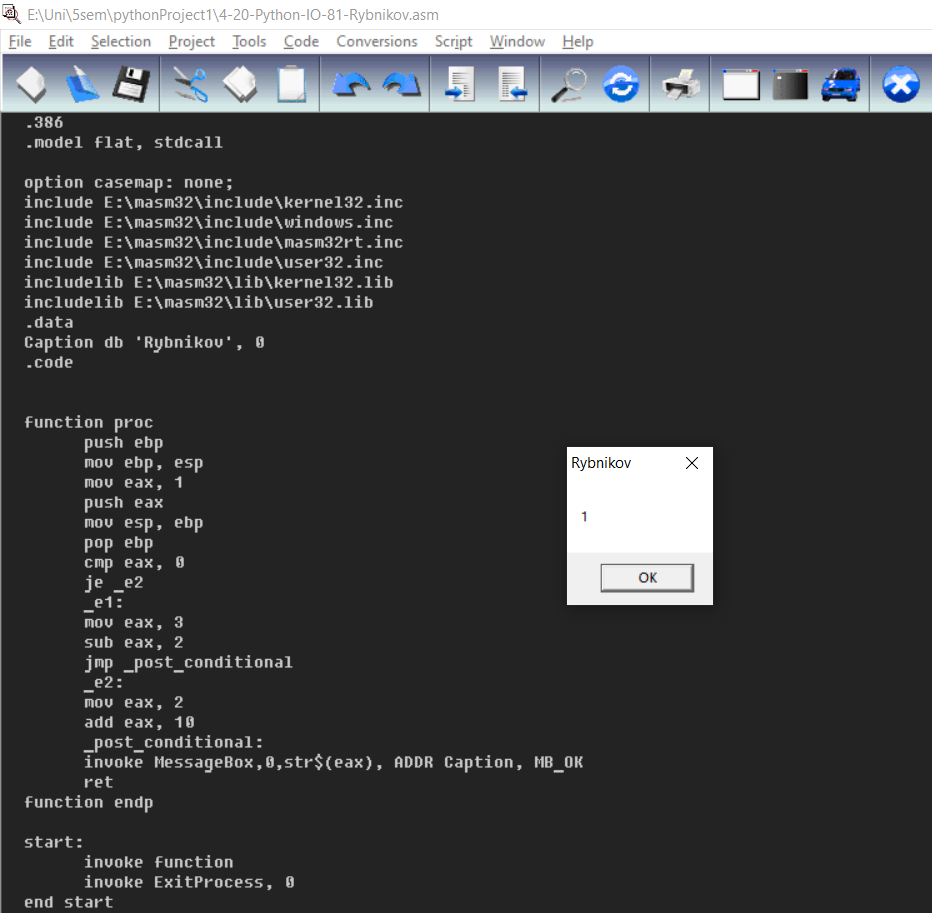
Вхідний код:



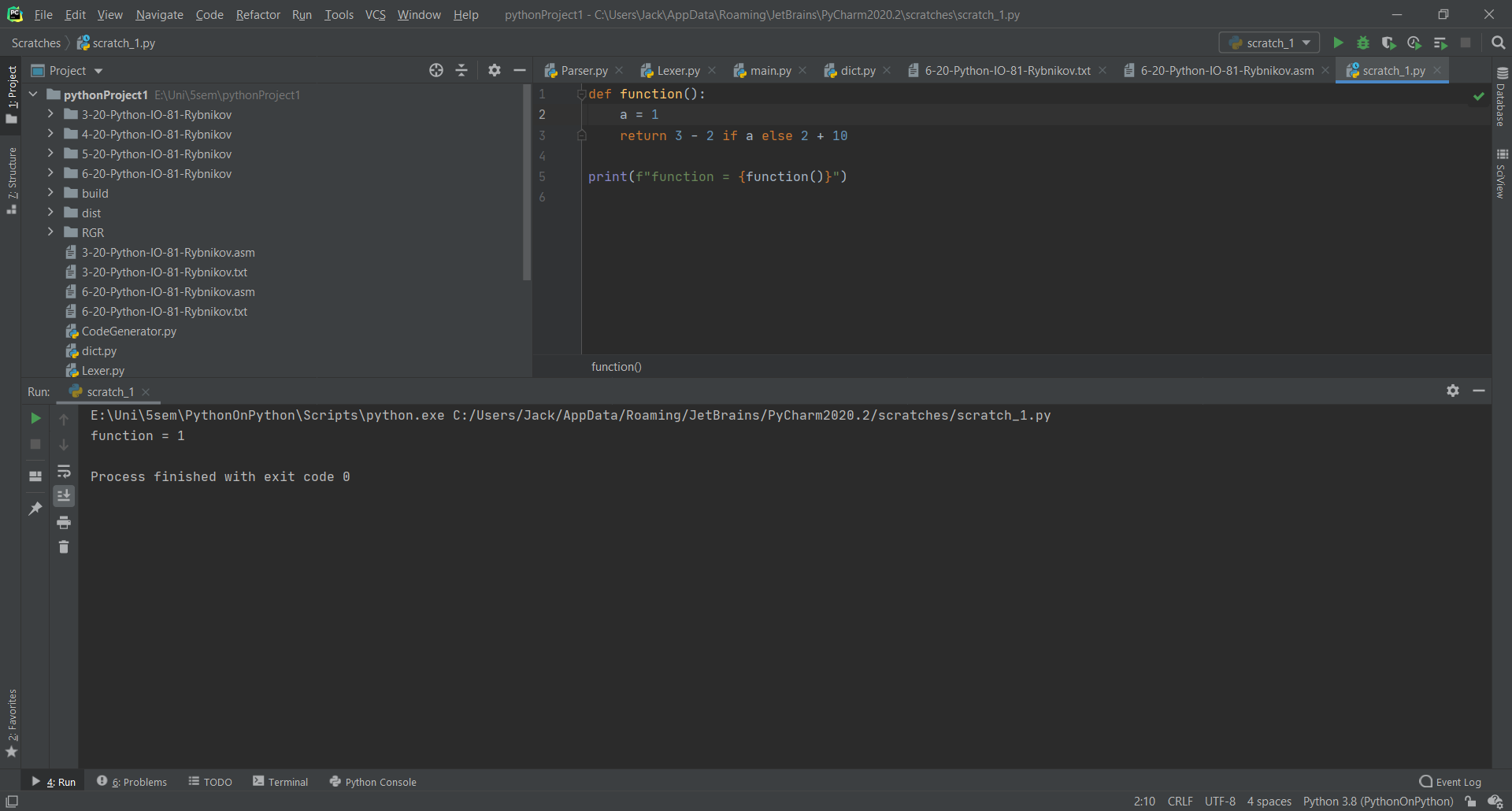
Вихідний код:



Перевірка роботи вихідного коду:

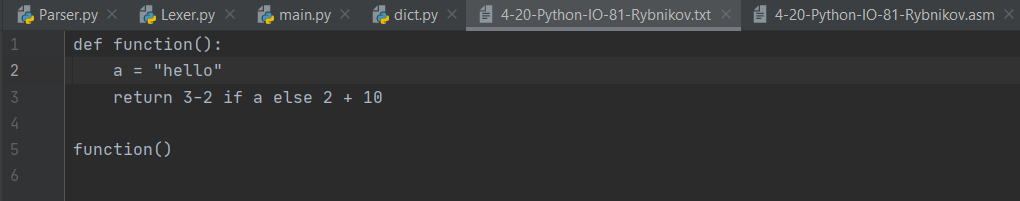


**Перевірка правильності розрахунків на мові Python:**

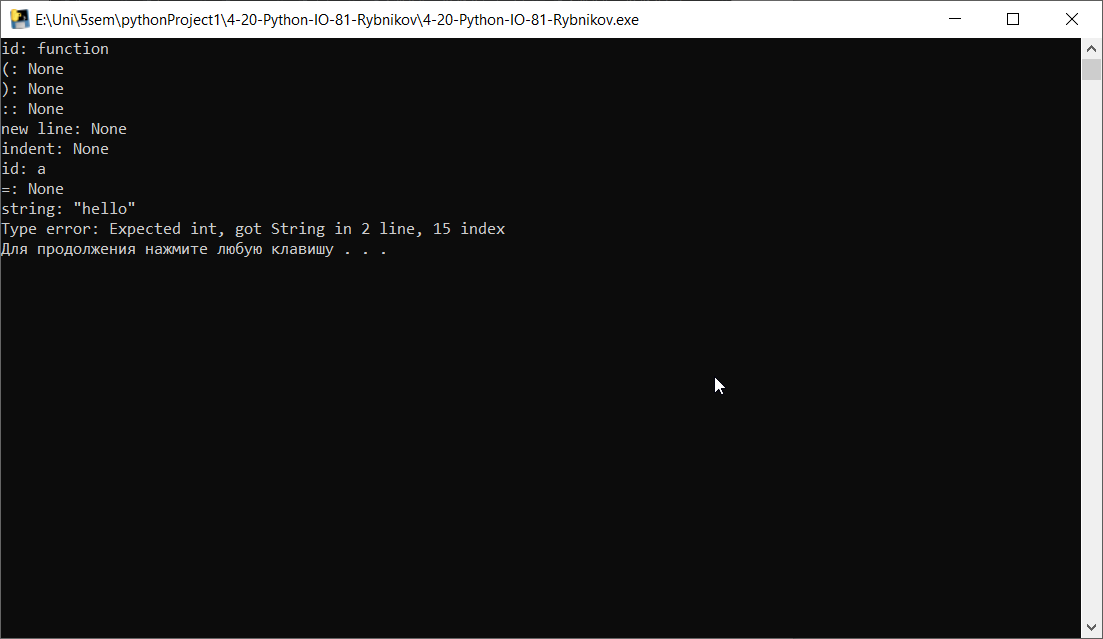


**Приклад 5**

Вхідний код:

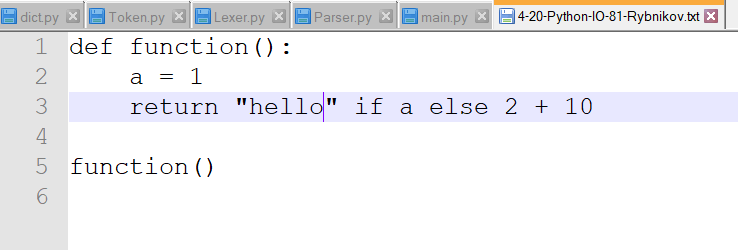


Вихідний код:

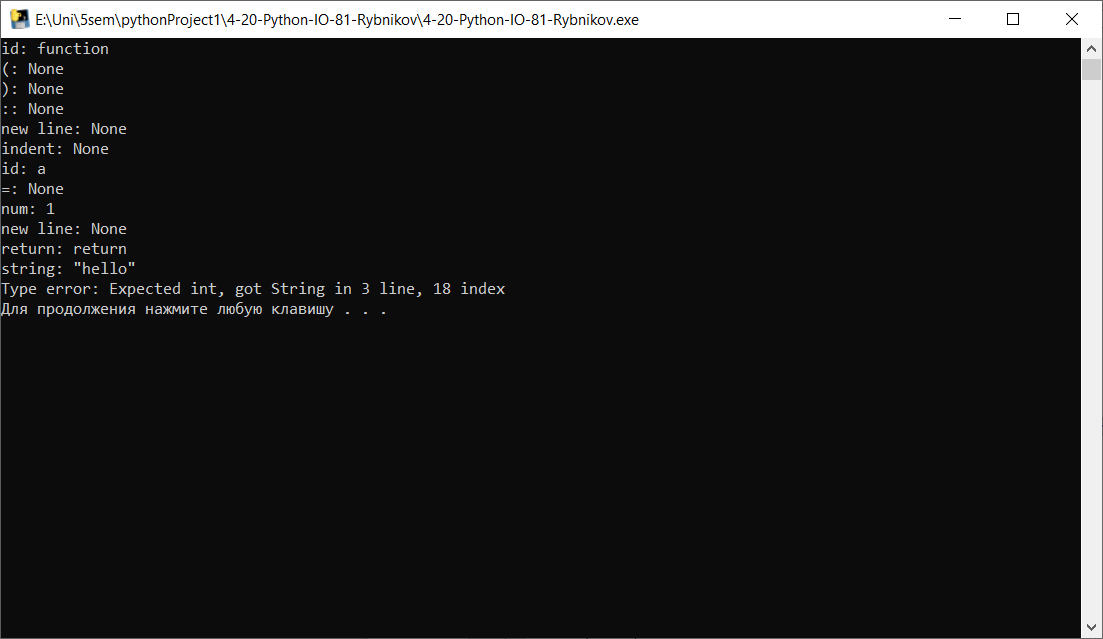


Приклад 6

Вхідний код:

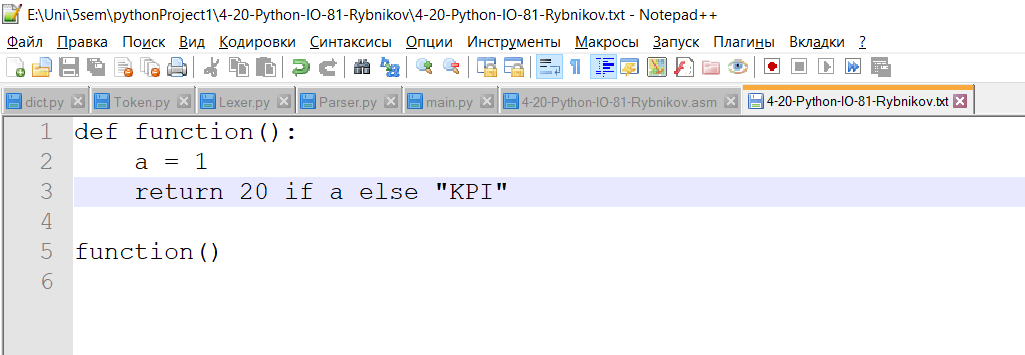


**Вихідний код:**

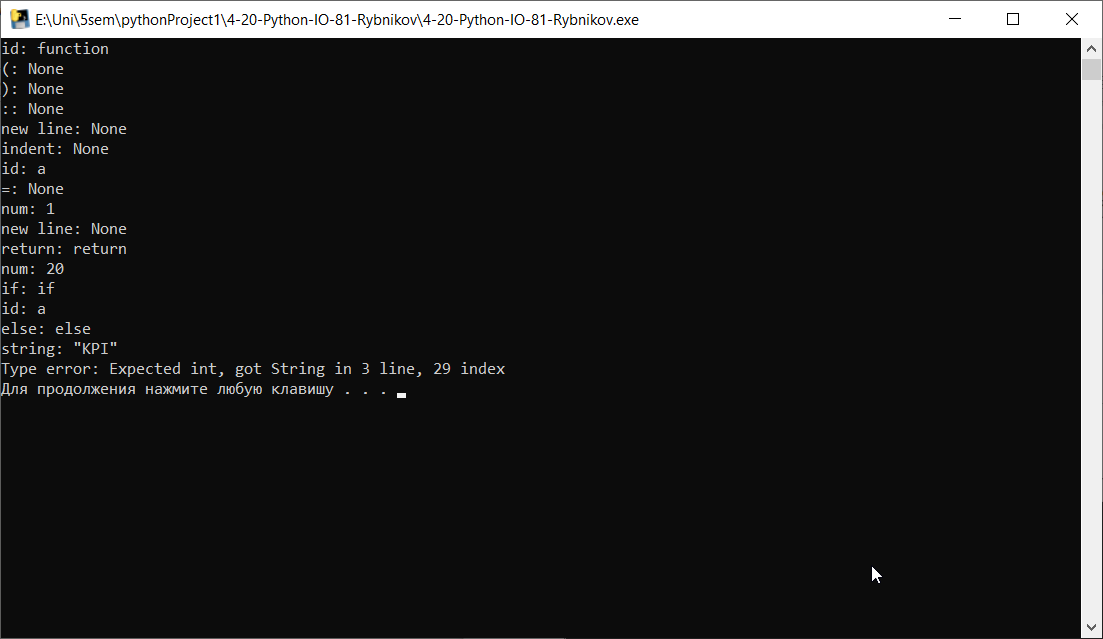


Приклад 7

Вхідний код:



**Вихідний код:**

****

**Висновки**

Було реалізовано конструкцію тернарного оператора мови Python. Програму було протестовано на контрольних прикладах, результати виявилися правильними.

**Контрольні питання**

**1. Що означає «встановити біт» (set a bit)?**

Коли ми встановлюємо біт, ми робимо його значення 1. Він виступає у ролі прапора.

**2. Чому використовується прапорець SF?**

SF — Sign Flag або прапор знаку.

**3. Що таке short-circuit evaluation і чому це важливо чи не важливо?**

Short-circuit evaluation використовується у булевих операціях для виразів, щоб коли умова виконалась не робити “зайві” розрахунки тим самим зменшуючи час обчислення і роблячи програми більш швидкодійними.

**4. В чому різниця між je та jmp ?**

jmp – безумовний перехід, je – умовний перехід(jump if equal).

**5. Що таке область видимості і в чому їх відмінності в різних мовах**?

Це ділянка програми, де існує окрема множина імен, пов'язаних із певними даними (змінними) чи оголошеннями (функціями, класами тощо). Однакові імена можуть вживатися на різних ділянках програми (а отже мати різні межі дії), але мати різне значення (посилатися на різні дані) чи стосуватися різних оголошень. У мові Python немає областей видимості як таких – ми можемо доступатись до змінних з будь-якого блоку, а також мати дві однакові змінні у різних або однакових блоках, змінна, яка знаходиться нижче перезапише змінну вище.

**6. Як працюють області видимості?**

У мові С ми можемо доступитись до змінної яка знаходиться у зовнішній області з внутрішньої, але не навпаки.

**7. Навіщо введені поняття block item та block?**

Бо оголошення змінної не statement, а expression, саме для цього нам потрібно окремо винести оголошення змінних.

**8. Як обробляється доступ до «правильної» змінної?**

Для кожного блоку створюється таблиця змінних. В результаті не будуть виникати конфлікти із зовнішнім блоком.

**9. У чому різниця парсингу між *conditional statements* та *conditional expressions*?**

В statement є тіло, в якому можна помістити багато інших statements.

В expression всередині може бути тільки інший expression.

**10. Як умовні оператори та вкладені конструкції відрізняються у мовах С та Python?**

В мові С вкладена конструкція – це частина коду, огорнута фігурними дужками. У мові Python вкладені блоки коду виділяються зсувом на задану кількість пробілів або простим табом. Закінчується блок зсувом вправо на ту ж саму кількість пробілів або табуляцією відносно останньої інструкції.

У мові С немає явно заданої конструкції else-if. У Python є конструкція elif.