Mērnieka uzdevums

4.izcilības (desmitnieka) uzdevums

Realizēt mērnieka uzdevumu - aprēķināt patvaļīga daudzstūra laukumu, ja zināmas tā malu garumu un leņķi starp malām.

Kods:

```
# Programmas nosaukums: Mērnieka uzdevums

# 4.izcilības (desmitnieka) uzdevums

# Uzdevuma formulējums: Realizēt mērnieka uzdevumu - aprēķināt patvaļīga daudzstūra laukumu, ja zināmas tā malu garumu un leņķi starp malām.

# Programmas autors: Vladislavs Babaņins

# Versija 1.0



import math


"""

Tika paņemta klase ComplexNumbers no 2.uzd MPR13.

Tāpēc lielāka daļa metožu netiek izmantota.

"""




class ComplexNumber:

    # Kompleksu skaitļu klase.

    def __init__(self, re=0, im=0):

        # Pēc noklusējuma izveido tukšu komplēksa skaitli (0 + 0i)

        # Ja ir norādots citādi, tad izveido tā, ka ievadīja lietotājs.

        self.re = re

        self.im = im
```

```
    def __repr__(self):
        # print()
        # Kompleksā skaitļu izvadīšanai lietotājam.
        if self.re != 0:  # Ja ir kāda reāla daļa, tad izvadam komplēkso skaitli ar realu daļu (neviss 0 + i*n)

            if self.im > 0 and self.im != 1:  # Ja imagināra daļa nav 1 un tā ir lielāka par 0, tad rakstām n + i, neviss n + k*i
                return f"{self.re} + {self.im}i"

            elif self.im == 1:  # Ja imagināra daļa ir 1, tad rakstām n + i, neviss n + 1*i
                return f"{self.re} + i"

            elif self.im == 0:  # Ja imagināra daļa ir 0, tad rakstām tikai reālu daļu n
                return f"{self.re}"

            elif self.im == -1:  # Ja imagināra daļa ir -1, tad rakstām n - i, neviss n - 1*i
                return f"{self.re} - i"

            else:  # Citā gadījumā rakstām n - k*i
                return f"{self.re} - {-self.im}i"

        else:  # Ja nav reālas daļas, tad nav jēgas rakstīt 0 + k*i, tad izvadam komplēkso skaitli tikai ar imagināru daļu k*i
            if self.im > 0 and self.im != 1:  # Ja imagināra daļa ir pozitīva un nav viens, tad izvadam k*i, neviss 0 + k*i
                return f"{self.im}i"

            elif self.im == 1:  # Ja imagināra daļa ir 1, tad izvadam i, neviss 0 + 1*i
                return "i"

            elif self.im == 0:  # Ja imagināra daļa ir 0, tad izvadam 0, neviss 0 + 0*i
```

```
            return "0"

        elif self.im == -1:  # Ja imagināra daļa ir 1, tad izvadam -i, neviss 0 - 1*i
            return "-i"

        else:  # Citādi izvādam -k*i (nav realas daļas un imagināra daļa ir negatīva un nav -1)
            return f"-{-self.im}i"

    def arg(self):
        # Atgriež komplēksa skaitļa argumentu.
        return math.atan2(self.im, self.re)

    def __add__(self, other):
        # +
        # Atgriež komplēksa skaitļa summu (self + other).
        real_sum = self.re + other.re
        imaginary_sum = self.im + other.im
        return ComplexNumber(real_sum, imaginary_sum)

    def __iadd__(self, other):
        # +=
        # Atgriež komplēksa skaitļa summu (self + other), bet kā __iadd__ (+=).
        # Atgriež jau eksistējošu mainīgu, neviss izveido jaunu mainīgu.
        self.re += other.re
        self.im += other.im
        return self

    def __sub__(self, other):
        # -
        # Atgriež komplēksa skaitļa starpību (self - other).
        real_diff = self.re - other.re
```

```
        imaginary_diff = self.im - other.im
        return ComplexNumber(real_diff, imaginary_diff)

    def __isub__(self, other):
        # -=
        # Atgriež komplēksa skaitļa summu (self - other), bet kā __isub__ (-=).
        # Atgriež jau eksistējošu mainīgu, neviss izveido jaunu mainīgu.
        self.re -= other.re
        self.im -= other.im
        return self

    def __mul__(self, other):
        # *
        # Atgriež komplēksa skaitļa reizinājumu (self * other).
        real_product = (self.re * other.re) - (self.im * other.im)
        imaginary_product = (self.re * other.im) + (self.im * other.re)
        return ComplexNumber(real_product, imaginary_product)

    def __imul__(self, other):
        # *=
        # Atgriež komplēksa skaitļa reizinājumu (self * other) bet kā __imul__ (*=).
        # Atgriež jau eksistējošu mainīgu, neviss izveido jaunu mainīgu.
        re1 = self.re
        im1 = self.im
        self.re = (re1 * other.re) - (im1 * other.im)
        self.im = (re1 * other.im) + (im1 * other.re)
        return self

    def __truediv__(self, other):
        # /
        # Atgriež komplēksa skaitļa dalījumu (self / other).
```

```
        denominator = (other.re * other.re) + (other.im * other.im)
        real_quotient = ((self.re * other.re) + (self.im * other.im)) / denominator
        imaginary_quotient = ((self.im * other.re) - (self.re * other.im)) / denominator
        return ComplexNumber(real_quotient, imaginary_quotient)

    def __itruediv__(self, other):
        # /=
        # Atgriež komplēksa skaitļa dalījumu (self / other) bet kā __itruediv__ (/=).
        # Atgriež jau eksistējošu mainīgu, neviss izveido jaunu mainīgu.
        denominator = (other.re ** 2) + (other.im ** 2)
        real_quotient = ((self.re * other.re) + (self.im * other.im)) / denominator
        imaginary_quotient = ((self.im * other.re) - (self.re * other.im)) / denominator
        self.re = real_quotient
        self.im = imaginary_quotient
        return self

    def __abs__(self):
        # Atgriež komplēksa skaitļa moduli.
        return math.sqrt(self.re * self.re + self.im * self.im)

    def conjugate(self):
        # Atgriež komplēksa skaitļas komplēksa saistīto skaitli.
        return ComplexNumber(self.re, -self.im)

    def __pow__(self, power):
        # Atgriež komplēksa skaitļi, kurš tika pacēlts naturāla pakāpe.
        modulus = self.__abs__() ** power
        arg = power * self.arg()
        re = modulus * math.cos(arg)
        im = modulus * math.sin(arg)
        return ComplexNumber(re, im)
```

```
def complex_power(z, n):
    # Atgriež komplēksa skaitļi, kurš tika pacēlts pakāpe.
    r = math.sqrt(z.re**2 + z.im**2)
    theta = math.atan2(z.im, z.re)
    re = r ** n * math.cos(n * theta)
    im_part = r ** n * math.sin(n * theta)
    return ComplexNumber(re, im_part)


def n_roots(self, n):
    # Atgriež sarakstu, ar visiem komplēksa skaitļa saknēm.
    # n - kuru sakni gribām izvilkt
    roots = []
    modulus = abs(self)
    arg = self.arg()
    for k in range(n):
        root_argument = (arg + 2 * k * math.pi) / n
        re = modulus * math.cos(root_argument)
        im_part = modulus * math.sin(root_argument)
        roots.append(ComplexNumber(re, im_part))
    return roots


def trigonometric_form(self):
    # Izvadīt lietotājam komplēksu skaitli trigonometriskajā formā.
    r = abs(self)
    theta = self.arg()
    return f"{r:.2f}(cos({theta:.2f}) + isin({theta:.2f}))"


def exponent_form(self):
    # Izvadīt lietotājam komplēksu skaitli eksponenciāla formā.
    modulus = abs(self)
```

```
        arg = self.arg()
        return f"{modulus} * e^({arg}i)"

    def exp(self):
        # Trigonomētriska formā
        re = math.cos(self.im)
        im = math.sin(self.im)
        return ComplexNumber(re, im)
```

'''

Šī programma aprēķina daudzstūra laukumu (daudzstūris ir bez šķersojumiem), ņemot vērā tā malu garumus un leņķus starp atbilstošiem malas garumiem.

Ir svarīga malas-leņķu secība, jo tikai tad var definētu vienu vienīgu daudzstūri un aprēķināt tam laukumu.

1. convert_angles_to_radians:

Šī funkcija pārvērš leņķus grādos radiānos. Pēc tam tas pielāgo leņķus, atņemot 180 grādus, lai iegūtu "ārejus" leņķus.

2. area_of_a_polygon_using_shoelace:

Šī funkcija aprēķina daudzstūra laukumu, izmantojot Gausa formulu (Shoelace formula).

Tas aprēķina virsotņu pāru "šķērsreizinājumu" un summē tos, lai aprēķinātu laukumu.

Tādu programmu veidojam praktiskas nodarbības laikā, kur pēc koordinātam varējam aprēķināt daudzstūra laukumu.

3. calculate_polygon_area:

Šī funkcija ir galvenā funkcija, kas darbojas ar daudzstūra malu sarakstu un leņķu sarakstu.

Tas pārvērš leņķus radiānos un izmanto tos, lai aprēķinātu daudzstūra virsotnes, attēlojot katru malu kā kompleksu skaitli tāda formā,

t.i., r * e^(i*theta), kur "r" ir malas garums. un "teta" ir leņķis līdz šai pusei.

Pēc tam tas aprēķina daudzstūra laukumu, izmantojot Gausa formulu.

Lietotajs ievada daudzstūra malas un leņķus, pēc tam programma aprēķina un izdrukā daudzstūra laukumu.

Faktiski ļoti līdzīgi koordinātu metodei, bet tikai ar komplēksa skaitļiem uz komplēksa plaknes.

Programma nepārbauda vai tas ievadītais daudzstūris reāli eksistē!

'''

```
def convert_angles_to_radians(angles):
    # Šī funkcija pārvērš visus dotos leņķus sarakstā no grādiem radiānos un atņem pi, lai pēc tam varētu strādat ar komplēksa skaitliem.
    # Atņem 180 grādi (pi) jo tad mēs dabūjam "pagriezienu" no Ox asi, jo leņķi skaita no Ox ass, un komplēksa skaitļiem tas būtu noderīgi.
    # Konvertē sarakstu ar leņķim grādos, ļeņkos radiānos un atņemam no visiem ļeņķiem pi.
    # Atgriež sarakstu ar konvertētiem ļeņkiem radianos no kuriem tika atņemta pi.
    # angles - saraksts ar leņķiem.

    converted_angles = []  # Izveidojam tukšu sarakstu, lai saglabātu konvertētos leņķus.
    for angle in angles:  # Ejam cauri katram leņķi no saraksta.
        adjusted_angle = angle - 180  # Pielāgojam leņķi, atņemot no tā 180 grādu.
        radians = adjusted_angle * math.pi / 180  # Pārvēršam pielāgoto leņķi radiānos.
        converted_angles.append(radians)  # Pievienojam konvertēto leņķi konvertēto leņķu sarakstam.

    return converted_angles  # Atgriežam konvertēto leņķu sarakstu.


def area_of_a_polygon_using_shoelace(vertices):
    # Aprēķina daudzstūra laukumu, izmantojot Gausa formulu (Shoelace formula).
    # Atgriež (area) laukumu daudzstūrim, kuram nav šķērsojumu.
    # vertices - saraksts ar visam virsotnēm "koordinātam" komplēksu skaitļu formā.
```

```
    area = 0  # Izveidojam mainīgu, kur glabāsies laukums. Izveidojam to kā nulle.

    for i in range(len(vertices)):  # Ejam cauri katrai virsotnei sarakstā.
        # Aprēķinām secīgu virsotņu "šķērsreizinājumu", (strādājam pēc formulas).
        vertex_i = vertices[i]  # Ņemam pašreizējo i virsotni.
        vertex_i_minus_1 = vertices[i - 1]  # Ņemam iepriekšējo i-1 virsotni.
        cross_product = vertex_i_minus_1.re * vertex_i.im - vertex_i.re * vertex_i_minus_1.im
# Pēc Gausa formulas.

        # Pievienojam daļēji izreķinātu laukumu, kopējam laukumam.
        area = area + cross_product

    # Kad cikls beidzam, tad dalam to izrēķinātu laukumu uz pusēm.
    area = 0.5 * area

    return area  # Atgriež aprēķināto laukumu.


def calculate_polygon_area(sides, angles):
    # Aprēķina nešķeršojušu daudzstūra laukumu, izmantojot komplēksus skaitļus.
    # sides - saraksts ar visiem daudzstūra malas garumiem (nosacītas vienības).
    # angles - saraksts ar visiem daudzstūra leņķiem grādos.

    # Konvertējam leņķus radiānos un atņemam no visiem pi.
    angles = convert_angles_to_radians(angles)

    # Izveido pirmo virsotni punkta 0,0 komplēksa plakne.
    vertices = [ComplexNumber(0, 0)]  # Punkts 0,0 plaknes vidu.
    theta = 0
    # Aprēķināsim virsotnes "koordinātes" komplēksa formā, lai pēc tam varētu izmantot
Gausa formulu.
    for i in range(len(sides)):  # Ejam cauri katrai malai sarakstā
```

```
        # Konvertēsim formā: r * e^(i*theta)
        theta += angles[i]  # Palielinām teta par pašreizējo i-to leņķi.

        # Konvertējam pašreizējo i-to malu par kompleksu skaitli un reizinām ar e^(i*theta).
        side_complex = ComplexNumber(sides[i], 0) * ComplexNumber(0, theta).exp()

        vertices.append(vertices[-1] + side_complex)  # Pievienojam virsotņu sarakstam nākamo
virsotni.

    # Kad cikls pabeidzies un visam virsotnēm tagad ir koordinātas,
    # (kā komplēksa skaitlis kur reāla daļa ir x koordināta, bet imagināra daļā ir y koordināta)
    # Aprēķinām laukumu, izmantojot Gausa formulu (Shoelace formula).
    area = area_of_a_polygon_using_shoelace(vertices)

    return abs(area)  # Atgriež aprēķinātā laukuma absolūto vērtību (drošības pēc).


# ---------------------------------------------------------
# Galvenā programmas daļa
# ---------------------------------------------------------


'''
Programma nepārbauda vai tas ievadītais daudzstūris reāli eksistē!
'''

num_sides = int(input("Ievadiet daudzstūra malas skaitu ==> "))

sides = []
angles = []
```

```
for i in range(num_sides):
    side_length = float(input(f"Ievadiet garumu {i+1}.malai ==> "))
    sides.append(side_length)

for i in range(num_sides):
    angle = float(input(f"Ievadiet {i+1}.leņķi grādos ==> "))
    angles.append(angle)

print("\nIevadīta daudzstūra laukums:")
print("S =", calculate_polygon_area(sides, angles))

'''
# Testa piemēri.
print("Kvadrāts ar malas garumiem 1:")

sides = [1, 1, 1, 1]  # Kvadrāts.
angles = [90, 90, 90, 90]
print("S =", calculate_polygon_area(sides, angles))  # Laukums ir 1

print("\nRegulārs piecstūris ar malas garumiem 3.53:")
sides = [3.53, 3.53, 3.53, 3.53]  # Regulārs piecstūris ar malas garumiem 3.53
(https://www.mathsisfun.com/geometry/area-polygon-drawing.html)
angles = [108, 108, 108, 108]
print("S =", calculate_polygon_area(sides, angles))  # Laukums ir 21.4

print("\nRegulārs trījstūris ar malas garumiem 5.2:")
sides = [5.2, 5.2, 5.2]  # Regulārs trijstūris ar malas garumiem 5.2
angles = [60, 60, 60]
print("S =", calculate_polygon_area(sides, angles))  # Laukums ir 11.69

print("\nRegulārs astoņstūris ar malas garumiem 2.3:")  #
https://www.mathsisfun.com/geometry/area-polygon-drawing.html
```

```
sides = [2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3, 2.3]
angles = [135, 135, 135, 135, 135, 135, 135, 135]
print("S =", calculate_polygon_area(sides, angles))  # Laukums ir 25.47

print("\nTaisnleņķa trijstūris ar malas garumiem 5, 3, 4")
print("Leņķi ir 36.87, 53.13, 60 grādi:")
sides = [5, 3, 4]
angles = [36.87, 53.13, 90]
print(calculate_polygon_area(sides, angles))  # Laukums ir 6

# Definējiet daudzstūra malas un leņķus grādos.
# Patvaļīgs ieliekts daudzstūris, tas laukums tika uzzināts uzzimējot to šajā internet-lappuse:
# https://www.mathsisfun.com/geometry/area-polygon-drawing.html
print("\nPatvaļīgs ieliekts daudzstūris 6.36, 5.36, 4.98, 4.12, 3.67, 3.15")
print("Leņķi ir 119, 54, 205, 38, 255, 49 grādi:")
sides = [6.36, 5.36, 4.98, 4.12, 3.67, 3.15]
angles = [119, 54, 205, 38, 255, 49]
print(calculate_polygon_area(sides, angles))  # Laukums ir 37.8
'''
```

Testa piemēri:

1)

```
Ievadiet daudzstūra malas skaitu ==> 3
Ievadiet garumu 1.malai ==> 1
Ievadiet garumu 2.malai ==> 1
Ievadiet garumu 3.malai ==> 1
Ievadiet 1.leņķi grādos ==> 60
Ievadiet 2.leņķi grādos ==> 60
Ievadiet 3.leņķi grādos ==> 60

Ievadīta daudzstūra laukums:
S = 0.4330127018922194
```

2) TESTA PIEMĒRI

```
Regulārs piecstūris ar malas garumiem 3.53:
S = 21.438696840999057

Regulārs trījstūris ar malas garumiem 5.2:
S = 11.708663459165614

Regulārs astoņstūris ar malas garumiem 2.3:
S = 25.542379489907344

Taisnleņķa trijstūris ar malas garumiem 5, 3, 4
Leņķi ir 36.87, 53.13, 60 grādi:
5.999977669787184

Patvaļīgs ieliekts daudzstūris 6.36, 5.36, 4.98, 4.12, 3.67, 3.15
Leņķi ir 119, 54, 205, 38, 255, 49 grādi:
37.87804342419757
```

3)

```
Ievadiet daudzstūra malas skaitu ==> 4
Ievadiet garumu 1.malai ==> 2
Ievadiet garumu 2.malai ==> 2
Ievadiet garumu 3.malai ==> 2
Ievadiet garumu 4.malai ==> 2
Ievadiet 1.leņķi grādos ==> 90
Ievadiet 2.leņķi grādos ==> 90
Ievadiet 3.leņķi grādos ==> 90
Ievadiet 4.leņķi grādos ==> 90

Ievadīta daudzstūra laukums:
S = 4.0
```

4)

```
Ievadiet daudzstūra malas skaitu ==> 5
Ievadiet garumu 1.malai ==> 1
Ievadiet garumu 2.malai ==> 1
Ievadiet garumu 3.malai ==> 1
Ievadiet garumu 4.malai ==> 1
Ievadiet garumu 5.malai ==> 1
Ievadiet 1.leņķi grādos ==> 108
Ievadiet 2.leņķi grādos ==> 108
Ievadiet 3.leņķi grādos ==> 108
Ievadiet 4.leņķi grādos ==> 108
Ievadiet 5.leņķi grādos ==> 108

Ievadīta daudzstūra laukums:
S = 1.7204774005889671
```

5)

```
Ievadiet daudzstūra malas skaitu ==> 6
Ievadiet garumu 1.malai ==> 1
Ievadiet garumu 2.malai ==> 1
Ievadiet garumu 3.malai ==> 1
Ievadiet garumu 4.malai ==> 1
Ievadiet garumu 5.malai ==> 1
Ievadiet garumu 6.malai ==> 1
Ievadiet 1.leņķi grādos ==> 120
Ievadiet 2.leņķi grādos ==> 120
Ievadiet 3.leņķi grādos ==> 120
Ievadiet 4.leņķi grādos ==> 120
Ievadiet 5.leņķi grādos ==> 120
Ievadiet 6.leņķi grādos ==> 120

Ievadīta daudzstūra laukums:
S = 2.598076211353316
```