Tentamen Programmeringsteknik I (Python) 2019-10-25
Lärare: Johan Öfverstedt

Skrivtid: 08:00 - 13:00

Tänk på följande:

- Skriv läsligt. Använd inte rödpenna.
- Skriv bara på framsidan av varje papper.
- Lägg uppgifterna i ordning. Skriv uppgiftsnummer (gäller B-delen) och din kod överst i högra hörnet på alla papper.
- Fyll i försättsbladet ordentligt.
- Såvida inget annat anges, både får och ska man bygga på lösningar från föregående uppgifter även om dessa inte har lösts.
- På B-delen är det tillåtet att införa hjälpfunktioner/hjälpmetoder och hjälpklasser. Uttrycket skriv en funktion/metod betyder alltså inte att lösningen inte får struktureras med hjälp av flera metoder.
- Du behöver inte skriva import-satser för att använda funktioner och klasser från standardbiblioteken.
- Alla uppgifter gäller programmeringsspråket Python och programkod skall skrivas i korrekt Python. Koden ska vara läslig med lämpliga variabelnamn, korta men beskrivande namn på funktioner, klasser och metoder.

  Observera att betyget påverkas negativt av
  - icke-privata variabler och onödiga variabler,
  - dålig läslighet,
  - upprepning av identisk kod,
  - underlåtenhet att utnyttja given/egen skriven kod.

Skrivningen består av två delar. Lösningarna till uppgifterna på A-delen ska skrivas in i de tomma rutorna och den delen ska lämnas in. Rutorna är tilltagna i storlek så att de ska rymma svaren. En stor ruta betyder inte att svaret måste vara stort! Lösningarna till uppgifterna på B-delen skrivs på lösa papper. För att bli godkänd (betyg 3) krävs att minst ca 75% av A-delen är i stort sett rätt löst. För betyget 4 krävs dessutom att minst hälften av uppgifterna på B-delen och betyg 5 att alla uppgifterna på B-delen är i stort sett rätt lösta. Vid bedömning av betyg 4 och 5 tas också hänsyn tillkvalitén på lösningarna i A-delen. Observera att B-delen inte rättas om inte A-delen är godkänd.

Lycka till!

Del A

A.1.

A)
`d = {}`
`d['a'] = 100`
`d['b'] = 42`
`print(d)`
	Vad skrivs ut?
	1) `{'a': 100, 'b': 42}`
	2) `{'b': 42}`
	3) `[100, 42]`
	4) `['a', 100, 'b', 42]`

B)
`def increment(x):`
`  x = x + 1`
`  return x`
`x = 1`
`y = increment(x)`
`print(x, y)`
	Vad händer?
	1) 2 2 skrivs ut
	2) 2 1 skrivs ut
	3) 1 2 skrivs ut
	4) 1 1 skrivs ut
	5) Det blir ett ValueError

C) Att testa om ett element finns i en samling `xs` innehållandes en miljon element med in-operatorn:
`if a in xs:`
`  ...`
är snabbast om?
	1) xs är en lista
	2) xs är ett lexikon (implementerat som en hash-tabell)
	3) xs är en tupel
	4) Alla datastrukturer är lika effektiva

D)
`def f(z):`
`  return 5 + z * 2`
`y = f(2)`
 	
Vad har variabeln `z` för datatyp?
	1) int
	2) float
	3) str
	4) list
	5) x har ingen bestämd datatyp

E) Vad är ett annat namn för `__init__`-metoden i klasser?	1) Main-metod 2) Generator 3) Skapare 4) Konstruktor

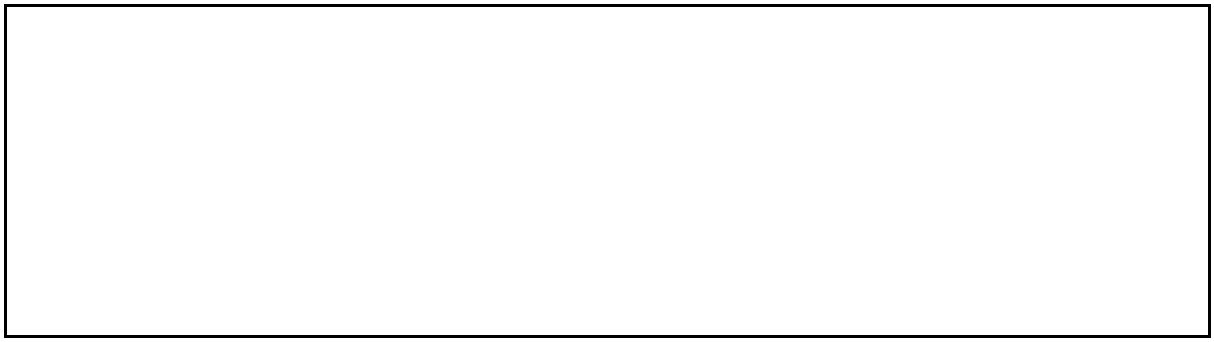
F)
`def f(x, y):`
`  return x + y`
`z = 5`
`w = 7`
`f(z, w)`
Vad är `z` och `w`?
	1) Argument
	2) Faktorer
	3) Attribut
	4) Parametrar

G)
`a = (1, 2, 'a')`
`b = (3, 4, 'b')`
`print(a + b)`
	Vad skrivs ut?
	1) `(4, 6, 'ab')`
	2) `(1, 2, 'a', 3, 4, 'b')`
	3) `[4, 6, 'ab']`
	4) Det blir fel

H) `a = [1, 2, 3, 4, 5]`
`b = a[-1::-2]`
`print(b)`
	Vad skrivs ut?
	1) `[2, 4]`
	2) `[5, 3, 1]`
	3) `[4, 2]`

A.2. Skriv en funktion `reverse_list` som givet en lista vänder på den. Funktionen ska inte returnera någonting, utan listan som skickas in ska ändras på plats. Ett sätt att göra detta på

är att byta plats på första och sista elementet, sen det näst första och näst sista, osv, i en loop.

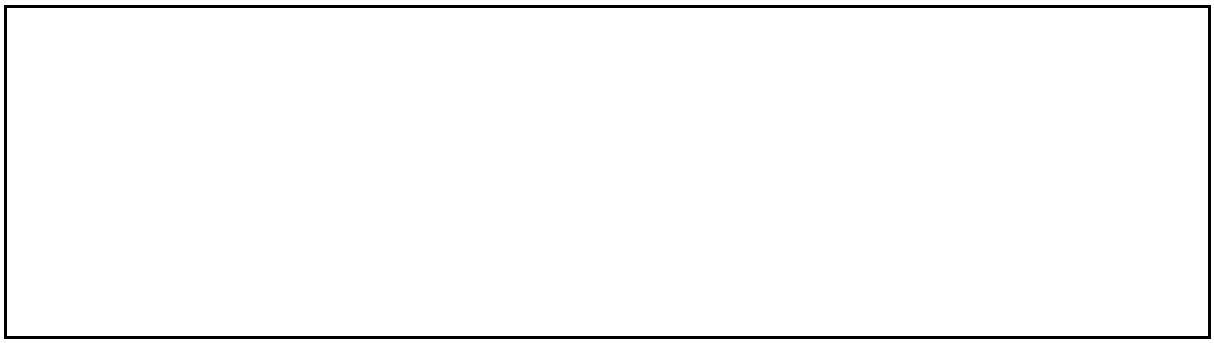


A.3. Skriv en funktion `factorial(m)` som beräknar och returnerar *n*-fakultet (!n) som defineras som n*(n-1)*(n-2)*...*1.

```
print(factorial(3))
print(factorial(4))
```

Ger utskrift:

```
6
24
```

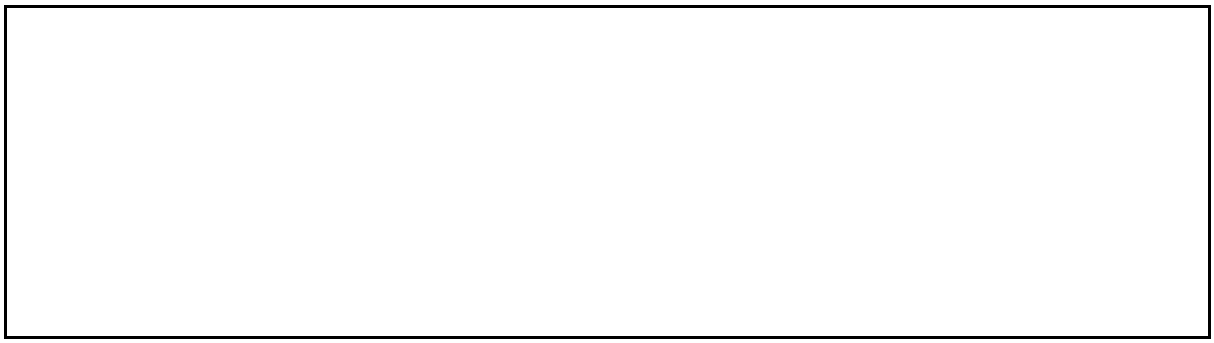


A.4. Skriv en funktion `maximum_with_index(lst)` som returnerar en tupel innehållandes det största elementet i listan lst och dess index `(value, index)`. Exempel:

```
print(maximum_with_index([1, 5, 2, 3, 7, 4]))
```

Ger utskrift:

```
(7, 4)
```



A.5. Skriv en funktion `letter_freq(s)` som skapar och returnerar ett lexikon med bokstavsfrekvenser för strängen s, där enbart tecken som är alfanumeriska inkluderas (detta kan testas med metoden str.isalpha()). Exempel:

```
print(word_freq('hej hej monica hej på dig monica'))
```

ger utskriften:

```
{'h': 3, 'e': 3, 'j': 3, 'm': 2, 'o': 2, 'n': 2, 'i': 3, 'c': 2,
'a': 2, 'p': 1, 'å': 1, 'd': 1, 'g': 1}
```

A.6. Skriv en funktion split(s, c), utan att använda inbyggda funktioner, som tar en sträng s och ett tecken c, och skapar en lista med strängar, där tecknet c markerar slutet på ett ord. Exempel:

print(split('hej hej monica hej på dig monica', ' ')

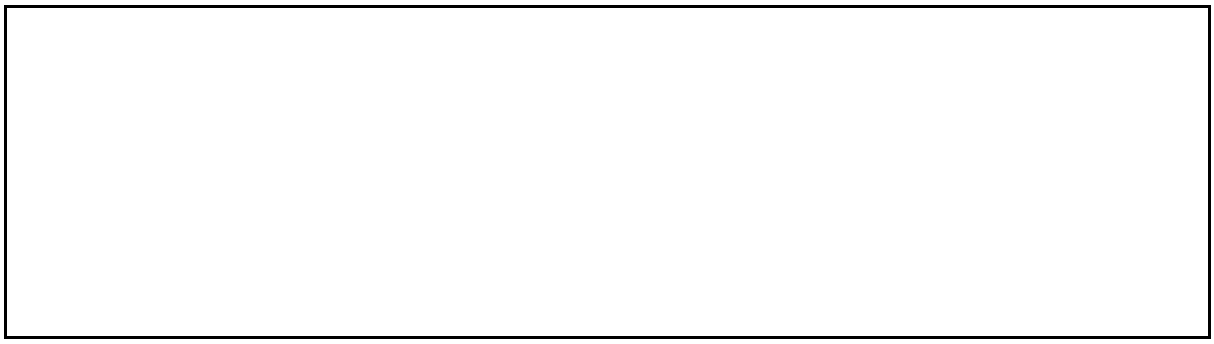
ger utskrift

['hej', 'hej', 'monica', 'hej', 'på', 'dig', 'monica']

De följande tre uppgifterna behandlar samtliga en klass Vector2D som representerar en tvådimensionell matematisk vektor. En tvådimensionell vektor kan representeras av ett par av tal `(x, y)`.

Givet två vektorer `(x_1, y_1)` och `(x_2, y_2)`.
Addition mellan två vektorer defineras som `(x_3, y_3) = (x_1 + x_2, y_1 + y_2)`.
Skalärprodukt mellan två vektorer defineras som `sp = x_1 * x_2 + y_1 * y_2`.

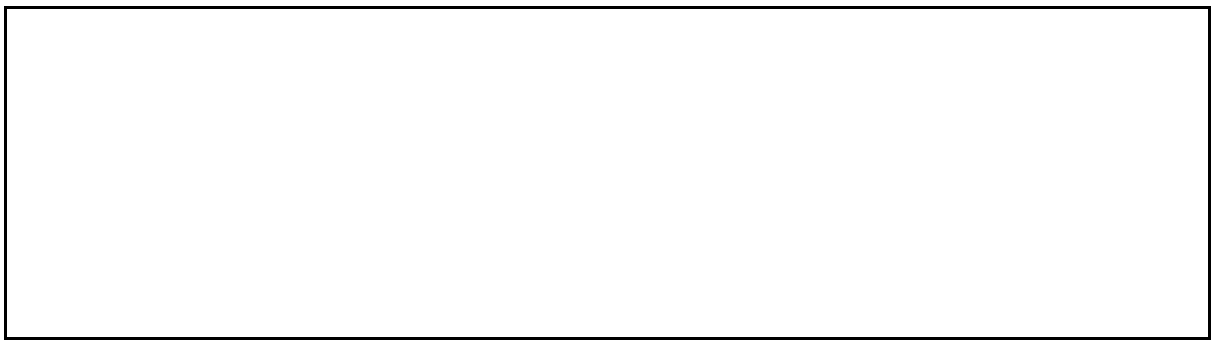
A.7. Definera klassen Vector2D och implementera `__init__(self, x, y)` som lagrar `x` och `y` som attribut.



A.8.

Implementera metoden __str__ i klassen Vector2D som returnerar en sträng som representerar vektorn.

Exempel:
v = Vector2D(2.0, 3.0)
print(v)
ger utskrift:
<2.0, 3.0>



A.9.

Implementera två metoder i klassen Vector2D:

`add(self, v)` som tar en annan vektor som parameter och returnerar den nya vektorn som resulterar av additionen.
`scalar_product(self, v)` som tar en annan vektor som parameter och returnerar skalärprodukten (ett heltal eller flyttal) mellan de två vektorerna.
Givet:

```
v1 = Vector2D(2.0, 3.0)
v2 = Vector2D(1.0, -2.0)
print(v1.add(v2))
```

ger utskrift:

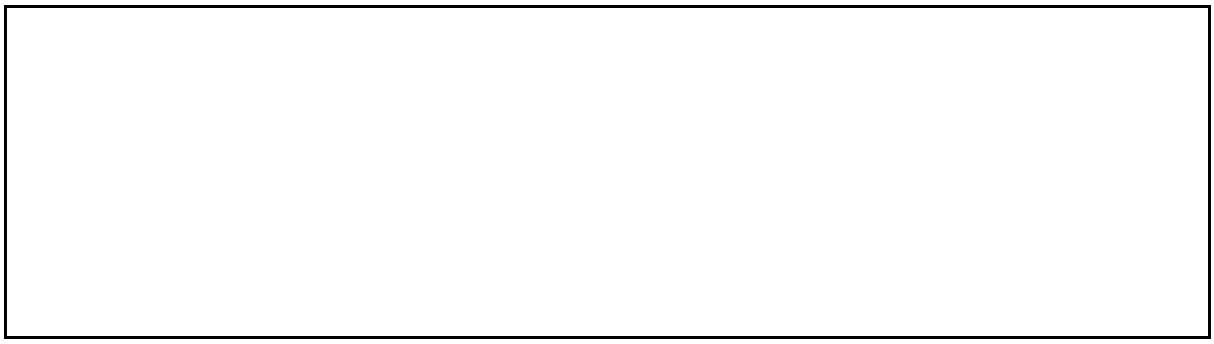
```
>> <3.0, 1.0>
```

och

```
print(v1.scalar_product(v2))
```

ger utskrift:

```
>> -4.0
```



A.10. Givet koden:

```
a = [(1, 2), (5, -7), (9, 3), (0, 1)]
b = sorted(a, key=f)
x1 = f((3, 4))
print(x1)
print(b)
```

ger utskrift:

```
>> 7
>> [(5, -7), (0, 1), (1, 2), (9, 3)]
```

Skriv nu funktionen `f` så som ger summan av elementen i den givna tupeln så att koden fungerar på så sätt att listan `a` sorteras med avseende på summan av elementen i tuplerna.

Del B

I denna del ska alla svar skrivas på lösa papper. Kom ihåg att skriva tentamenskod på varje blad. Använd ett blad per uppgift.

Följande program (i filen `snake.py`) är givet, vilket är en början på en implementering av ett litet Snake-spel.

```
import random

# Direction constants
LEFT = 0
DOWN = 1
RIGHT = 2
UP = 3

# Computes the new direction given an old direction and
# a 'turn' value
def compute_turn(direction, turn):
   assert(-1 <= turn <= 1)
   new_dir = direction + turn
   if new_dir == -1:
       new_dir = 3
   elif new_dir == 4:
       new_dir = 0
   return new_dir

# Computes the step vector given an input direction
def compute_step(dir):
   if dir == LEFT:
       return (-1, 0)
   elif dir == DOWN:
       return (0, 1)
   elif dir == RIGHT:
       return (1, 0)
   elif dir == UP:
       return (0, -1)
   else:
       raise ValueError(f'Illegal direction: {dir}')
```

```
d = {'L': 1, 'l': 1, 'F': 0, 'f': 0, 'R': -1, 'r': -1}

def main():
   random.seed(0)
   s = Snake(5, (3, 3))
   while True:
       print(s)
       t = input('Move (L, F, R): ')
       if t in d:
           if s.move(d[t]) == False:
               print('Game Over!')
               break
           if s.check_win():
               print('You Win!')
               break
       else:
           print('Illegal move')

if __name__ == '__main__':
   main()
```

Testutskrift vid en provkörning:

```
.....
.....
@....
...#.
.....

Move (L, F, R): f
.....
.....
@....
..#..
.....

Move (L, F, R): f
.....
.....
@....
.#...
.....

Move (L, F, R): f
.....
.....
@....
#....
.....

Move (L, F, R): r
.....
Move (L, F, R): r
.....
....@
...OO
....#
.....

Move (L, F, R): r
.....
....@
....O
...#O
.....

Move (L, F, R): r
.....
....@
...#.
...OO
.....

Move (L, F, R): r
.....
....@
...O#
...O.
.....

Move (L, F, R): l
```

```
.....
#....
O...@
.....

Move (L, F, R): r
.....
.....
O#...
....@
.....

Move (L, F, R): f
.....
.....
.O#..
....@
.....

Move (L, F, R): f
.....
.....
..O#.
....@
.....

Move (L, F, R): f
.....
.....
...O#
....@
.....
```

```
.....
..@.#
...OO
...O.
.....

Move (L, F, R): l
.....
..@#O
...OO
.....
.....

Move (L, F, R): f
.....
..#OO
...OO
.....
@....

Move (L, F, R): f
.....
.#OOO
....O
.....
@....

Move (L, F, R): f
.....
#OOOO
.....
.....
@....

Move (L, F, R): f
Game Over!
```

B.1. Definiera klassen `Snake` och skriv `__init__`-metoden som tar emot en spelplansstorlek `sz` (heltal) och en startposition i form av en tupel `(x, y)`, som förväntas vara inuti spelplanen: `0 <= x < sz`, och `0 <= y < sz`.

Klassen ska ha följande attribut: `sz` (spelplansstorleken), `snake` (en lista med ormens delars positioner i form av tupler `(x, y)`, som initialt bara innehåller startpositionen), direction som är ett heltal `0 <= direction <= 3`, som givet av konstanterna högst upp i det givna programmet, och `fruit_pos` som representerar fruktens (som ormen ska fånga för att växa, representerad av '@') position.

Antingen kan den initiala `fruit_pos` sättas till en slumpmässig plats med ett anrop till metoden `place_fruit` från uppgift B2, eller så kan den placeras på en godtycklig plats som inte kolliderar med ormens startposition.
B.2. Skriv en metod `place_fruit` som placerar ut frukten (ändrar `fruit_pos`) på en slumpvist vald ledig position `(x, y)` som inte krockar med ormen.

Tips: Ett sätt att lösa detta är att slumpa fram en position med två `random.randint` anrop, inuti en loop som fortsätter tills en giltig (ledig) position har hittats. Detta är inte den enda

möjliga eller bästa lösningen, men duger bra för den här tentamen.

Skriv även metoden `check_win` som returnerar true om spelet har vunnits, vilket sker om ormen fyller alla platser på spelplanen (som har `sz * sz` antal platser).

B.3. Skriv en metod `move(turn)` som givet en vridning representerad av heltalen -1: höger, 0: framåt, +1: vänster, flyttar fram ormen ett steg. Frukten ska tas om ormen kolliderar med den och ormen ska då växa. Om ormen går utanför spelplanen eller om den kolliderar med sig själv ska False returneras för att signalera att spelet är över. Om förflyttningen lyckades ska True returneras.

Följande logik ska implementeras, i ordning:

1) beräknar den nya riktningen med hjälpa av den givna funktionen `compute_turn`,
2) beräknar stegvektorn med hjälp av den givna funktionen `compute_step`,
3) beräkna den nya positionen på ormens huvud genom att addera stegvektorn till föregående position för huvudet,
4) lägg till ormens nya huvudposition till ormens lista,
5) om huvudet befinner sig på samma position som frukten, slumpa fram en ny position för frukten med hjälp av `place_fruit` från B.2,
6) annars, om inte 5 gjordes, ta bort den sista delen av ormens kropp (så att den verkar förflytta sig och bibehålla sin storlek) med metoden `pop` för ormens lista; testa om huvudets nya position krockar med ormens kropp, eller är utanför spelplanen (inte `0 <= x <= self.sz`, …), om så är fallet returnera False för att markera att spelet är förlorat. Annars returnera True.

B.4. Skriv metoden __str__ som returnerar en sträng som representerar spelplanen. Se testutskriften ovan som ett exempel på hur
1) ormens huvud (#),
2) ormens kropp (O),
3) frukten (@),
4) bakgrunden (.),
5) radbrytning ('\n')
ska se ut.

Tips: Bygg en lista med delsträngar och använd strängmetoden `join`.

Referensblad

Listor

`len(lst)` - Ger listans längd.

`sorted_list = sorted(lst)` - Sorterar elementen i listan och returnerar en ny lista. Den ursprungliga listan förändras inte.

`sorted_list = sorted(lst, key=f)` - Sorterar elementen i listan efter den nyckel som returneras av funktionen som ges till parametern `key`.

`lst.pop(index)` - Tar bort och returnerar elementet på plats `index`.

Exempel `['a', 'b', 'c'].pop(0)` ger `'a'`.

`lst.append(value)` - Lägger till ett nytt element sist i listan.

Strängar

`len(s)` - ger längden på given sträng.

`s.join(lst)` - sammanfogar en lista till en sträng med s mellan varje element.

Exempel `'-'.join('1', '2', '3')` ger `'1-2-3'`.

`s.count(subs)` - räknar antalet förekomster av strängen `subs` i strängen `s`.

Exempel `'abbabba'.count('bb')` ger `2`, `'abbabba'.count('b')` ger `4`.

s.split() - Delar upp en sträng till en lista med strängar, som separeras av whitespace (mellanslag, tab, radbrytningar, osv).

Exempel `'hej på dig'.split()` ger `['hej', 'på', 'dig']`

Sekvenser

`range(start, stop, step)` - skapar en sekvens med heltal från `start` till men inte med `stop` med steglängd `step`.

Exempel:

`list(range(7, 2, -1))` ger `[7, 6, 5, 4, 3]`.

`enumerate(lst, start=0)` - ger en sekvens med par (index, värde) från listan lst, där startindex ges av parametern start.

Lexikon

`len(lexikon)` - ger antalet nyckel/värde-par i ett givet lexikon.

```
for key, value in lexikon.items():
    # kod som använder nyckel och värde
```

hämtar nycklar och motsvarande värden från ett lexikon.

Slumptalsgenerering

`random.random()` - ger ett slumptal (flyttal) i intervallet [0.0, 1.0].

`random.randint(a, b)` - ger ett slumptal (heltal) i `{a, a+1, ..., b-1, b}` alltså inklusive startvärde och stoppvärde.

In/utmatning

`print(x, y, z, …, sep=', ', end=' ')` - skriver ut en serie uttryck ett efter ett, separerade av `sep`, och avslutas med `end`.

`input(msg)` - Visar meddelandet msg och läser in en sträng från användaren som returneras.