Tentamen Programmeringsteknik I (Python) 2019-12-17
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 = (1, 2, 3)`
`d[0] = 7`
`print(d)`
	Vad skrivs ut?
	1) Strängen `'d'`
	2) `(1, 2, 3)`
	3) `[1, 2, 3]`
	4) Det blir fel. Tupler är oföränderliga.

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

C) Python kan kategoriseras som ett:	1) Skriptspråk 2) Rent funktionellt språk 3) Rent objektorienterat språk 4) Ett naturligt språk

D)
`def f(z):`
`  return 5 + z * 2`
`y = f(2)`
 	
Vad har värdet i variabeln `y` för datatyp?
	1) int
	2) float
	3) str
	4) list
	5) Värdet i variabeln `y` 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 `x` och `z` av följande alternativ?
	1) x: Argument, z: Parameter
	2) x: Parameter, z: Argument
	3) x: Attribut, z: Attribut
	4) x: Datatyp, z: Datatyp

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

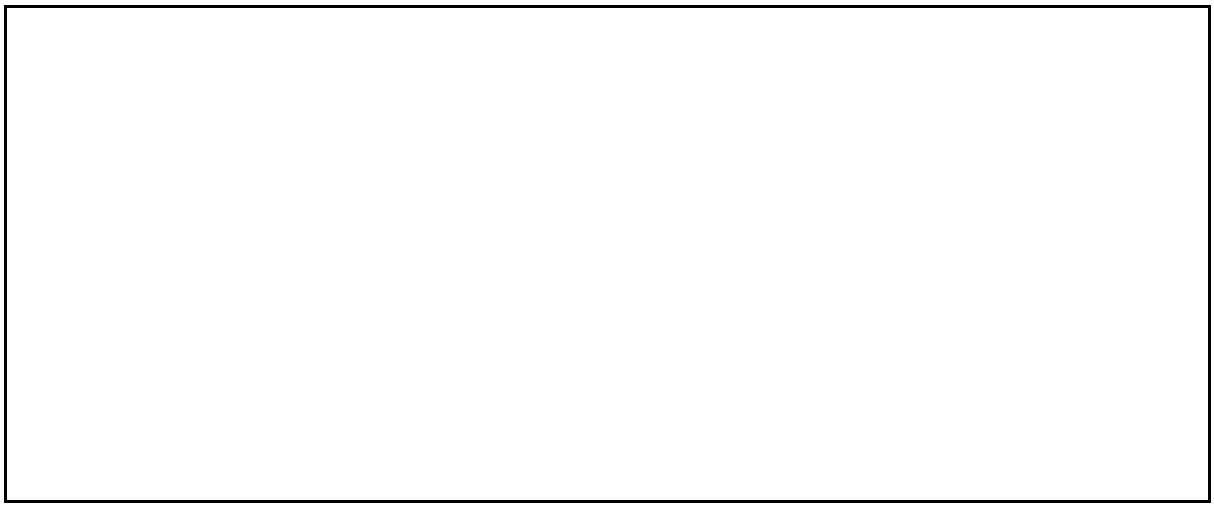
H) `a = [5, 4, 3, 2, 1]`
`b = a[-1::-2]`
`print(b)`
	Vad skrivs ut?
	1) `[2, 4]`
	2) `[5, 3, 1]`
	3) `[4, 2]`
	4) `[1, 3, 5]`

A.2. Skriv en funktion `add_two_lists(a, b)` som givet två listor adderar dem elementvis och returnerar en ny lista innehållandes resultatet. Om listorna innehåller olika många

element ska bara så många element som finns i den kortaste listan inkluderas i resultatet.

Exempel:

```
r=add_two_lists([1, 2, 3, 4, 5], [7, 8, 9])
print(r)
>> [8, 10, 12]
r=add_two_lists([5, 8], [3, 9, 1])
print(r)
>> [8, 17]
```



A.3. Skriv en funktion `fibonacci(n)` som beräknar och returnerar det n:te Fibonacci talet som defineras som `fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)`, där `fibonacci(0) = 0`, `fibonacci(1) = 1`, `fibonacci(2) = 1`.
Detta kan beräknas med en loop och två lokala variabler, a1, a0, upp till `n`:
atmp = a1 + a0
a0 = a1
a1 = atmp

```
print(fibonacci(3))
print(fibonacci(4))
print(fibonacci(5))
print(fibonacci(6))
```

Ger utskrift:

```
2
3
5
8
```

A.4. Skriv en funktion `median(lst)` som beräknar och returnerar medianen av listan. Om antalet element är jämnt, ska medelvärdet av de två mittenelementen (i en ordnad lista) returneras, och annars ska mittenelementet (i en ordnad lista) returneras. Exempel:

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

Ger utskrift:

```
3.5
3
```

A.5. Skriv en funktion `word_freq(s)` som skapar och returnerar ett lexikon med ordfrekvenser 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:

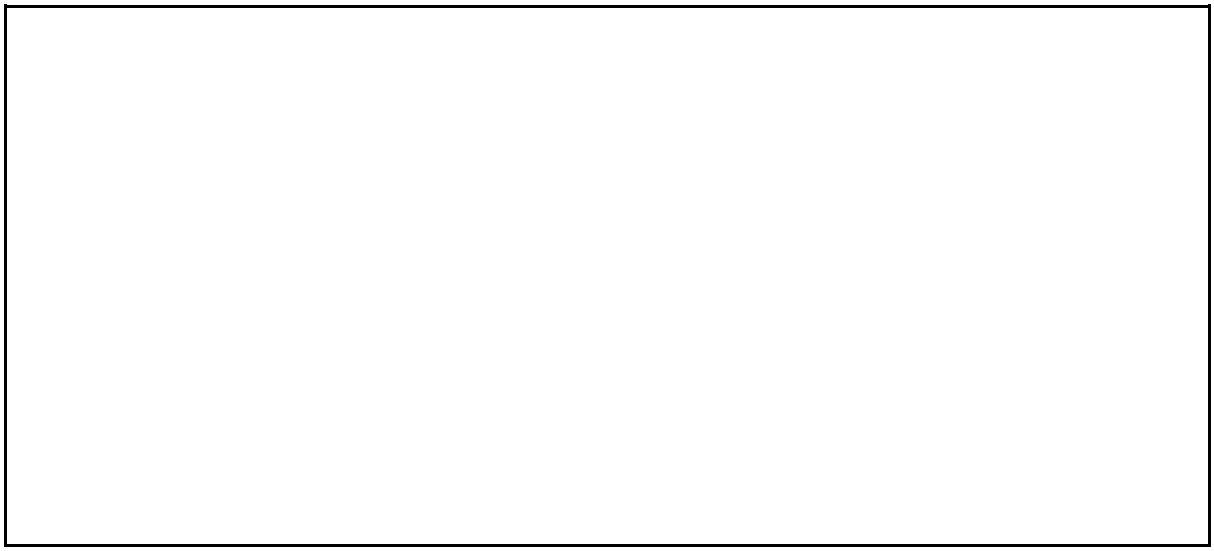
```
{'hej': 3, 'monica': 2, 'på': 1, 'dig': 1}
```

A.6. Definera en klass PokerDice som representerar fem tärningar (representerande de 13 olika valörerna i en pokerlek: 2, 3, 4, 5, 6, 7, 8, 9, 10, Kn, D, K, E). Tärningarnas värde bör lagras internt som en lista med heltal. Skriv klassens __init__-metod som skapar en initial uppsättning tärningsvärden.

A.7. Skriv __str__-metoden för klassen PokerDice, som givet en instans av dessa tärningar, returnerar en strängrepresentation för dem. Tips: Skapa en lista med alla valörer och använd tärningens heltalsvärde som ett index för att slå upp dess strängrepresentation. Exempel för slumpmässigt valda tärningar. __str__ ska givetvis fungera generellt och inte enbart för

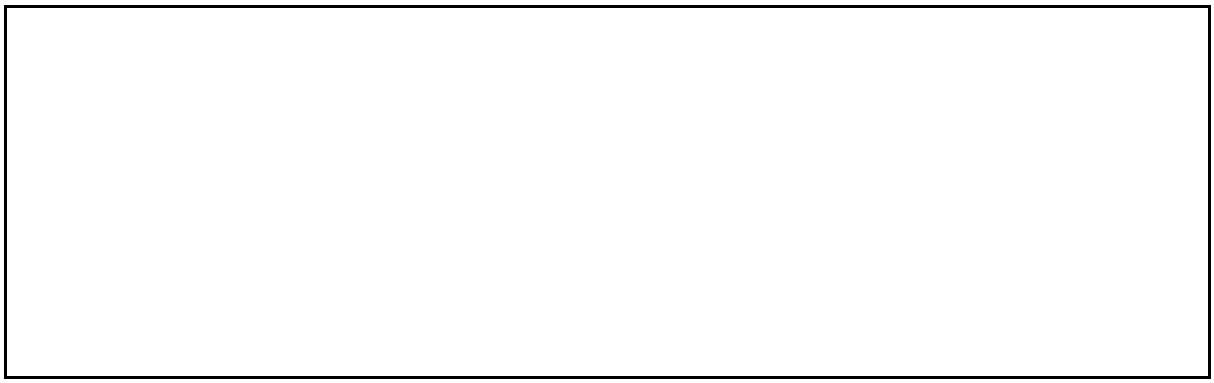
dessa exempelkonfigurationer:

```
print(d1)
>> [10, Kn, D, K, E]
print(d2)
>> [4, 2, 7, Kn, 8]
```



A.8. Implementera en metod `roll` i klassen PokerDice som rullar tärningarna slumpmässigt (ger en instans av klassen PokerDice en ny slumpmässiga tärningskonfiguration). Tips: `random.randint(min, max)` kan komma till nytta här.

Exempel:
print(d)
>> [4, 5, 6, 7, 8]
d.roll()
print(d)
>> [2, 5, 5, Kn, E]



A.9. Skriv en funktion bubblesort(lst) som sorterar listan lst med bubbelsortering, vilket är en ineffektiv men enkel metod för att sortera en lista.
Bubbelsortering fungerar på följande sätt:
För varje element, jämför med närmaste granne, och om de är i fel ordning (den

efterföljande är mindre än den föregående), byt plats på dem. Upprepa denna procedur i en loop tills inget element har bytt plats efter en fullständig genomgång av samtliga element.

A.10. Skriv en funktion `random_shuffle(lst)` som ger en slumpmässig blandning (permutation) av en lista. Ett sätt att göra detta är att för varje element e1, generera ett slumpmässigt heltalsindex för ett element e2, och låt e1 och e2 byta plats.

Kom ihåg att man kan slumpa heltal med `random.randint(min, max)`.

Exempel:

```
print(random_shuffle([1, 2, 3, 4, 5]))
>> [2, 3, 1, 5, 4]
print(random_shuffle([1, 2, 3, 4, 5]))
>> [3, 5, 2, 4, 1]
```

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.

Alla uppgifter i del B handlar om ett program som liknar de irrande sköldpaddorna.

Här ska vi simulera ett antal partiklar som rör sig slumpmässigt, med en maximal angiven steglängd. När en partikel vandrar utanför en given yta förstörs partikeln och en ny partikel skapas i dess ställe på en slumpmässig plats.

Varje tidssteg räknar programmet hur många partiklar som finns i varje ruta och ett rutnät med dessa räknare kan slutligen skrivas ut.

Huvudprogrammet ges av:

```
import random

... (Plats för koden som ska implementeras i B1-B4)

bm = BrownianMotion(5, 5.0, 16)

for _ in range(20):
    bm.move()
    bm.record()

print(str(bm))

for _ in range(1000):
    bm.move()
    bm.record()

print(str(bm))
```

Exempel på resultatet av en körning av programmet:

```
Particle recreated
Particle recreated
Particle recreated
Particle recreated
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 02 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 03 01 00 01 00 00 00 00 00 00 00
00 00 02 00 00 01 00 00 01 00 00 00 00 00 00 00
04 07 01 00 03 01 01 01 00 00 00 00 00 00 00 00
00 00 04 03 05 01 01 00 00 02 00 00 00 00 00 00
00 00 00 02 07 02 03 01 00 01 00 00 00 00 00 00
```

```
00 00 00 00 01 00 06 00 00 00 00 00 00 00 00 00
00 00 00 00 01 01 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 02 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 05 02 00 00 00 00 00 00
00 00 00 00 00 00 00 00 01 04 00 00 00 02 00 00
00 00 00 00 00 00 00 00 00 01 02 03 00 00 01 02
00 00 00 00 00 00 00 00 00 02 02 00 00 01 00 00
Particle recreated
Particle recreated
```
… (hoppar över många 'Particle recreated')
```
Particle recreated
Particle recreated
04 05 09 02 19 07 24 12 14 16 08 08 08 03 04 01
05 06 09 09 20 27 15 11 18 17 16 17 15 06 09 07
04 05 13 25 25 29 14 19 25 29 18 32 12 12 05 04
05 10 19 31 38 37 34 34 31 42 22 24 22 15 13 03
04 13 27 21 35 41 33 30 33 31 36 37 15 12 16 11
05 15 26 31 39 33 39 46 22 35 45 24 37 28 24 11
11 11 12 29 32 26 41 68 53 27 38 27 14 09 26 12
13 15 26 34 45 33 41 40 34 29 22 37 26 17 10 10
09 30 21 42 59 62 39 38 24 23 28 25 21 20 06 04
08 17 41 38 51 42 33 34 23 21 26 15 20 12 07 08
09 09 17 36 64 55 31 34 26 21 23 22 10 17 15 06
11 24 25 19 34 22 28 29 32 36 19 15 11 11 06 04
05 14 12 27 18 26 27 32 28 16 17 13 12 12 10 02
12 11 13 09 20 27 30 23 26 19 11 17 06 07 05 05
06 15 10 08 15 12 15 21 19 06 14 15 10 04 05 08
00 07 04 07 06 06 09 12 17 13 06 04 07 10 02 02
```

B1. Skriv två funktioner.

- `create_particle(sz)` som returnerar en tupel med 2 slumpvisa flyttalskoordinater inom intervallet `0 <= x < sz` och `0 <= y < sz`. Tips: använd random.random().
- `move_particle(p, step, sz)` som givet en tupel p (med två koordinater) och en längsta steglängd och en storlek sz för en begränsad yta, beräknar en ny tupel av koordinater givet ett steg från utsprungliga koordinater med en slumpvis steglängd mellan 0 och step. Om den nya positionen hamnar utanför rutnätet, så att `0 <= x < sz` och `0 <= y < sz` inte längre gäller, så ska en ny partikel skapas och returneras, och meddelandet 'Particle recreated' ska skrivas ut.

B2. Definera klassen BrownianMotion.
Skriv en __init__-metod med parameterarna:
count - antal partiklar
step - längsta möjliga steglängd
sz - storlek på rutnätet som ska användas för att hålla reda på antalet partiklar som befunnit sig i varje ruta.

Utöver att spara parametrarna som attribut, ska även ett tvådimensionellt rutnät (`sz` x `sz`) bestående av nästlade listor skapas (ursprungligen med värdet 0 som alla element).
Det angivna antalet partiklar, i form utav tupler med två element, ska skapas (med funktionen create_particle från B1) och lagras i en lista.

B3. Skriv två metoder i klassen BrownianMotion.

- `move(self)` som förflyttar alla partiklarna slumpmässigt med hjälp av funktionen move_particle från B1. Alla partiklarna i listan (som är ett attribut som skapades i B2) ska ersättas med sin nya position.
- `record(self)` som, för varje partikel, beräknar vilken heltalsruta den befinner sig i (med trunkering av decimalerna genom konvertering till heltal), och ökar på motsvarande rutas räknare i det tvådimensionella rutnät som skapades i B2, med `1`.

B4. Skriv en metod i klassen BrownianMotion: __str__.

Denna metod ska returnera en sträng som representerar det rutnät med räknare som kan ses i körexemplet ovan och som skapades i B2, och uppdaterades i metoden `record` från B3.

Tips: Använd f-strängar. Man kan formatera heltal med följande syntax så att alla tal representeras av minst två tecken, oavsett vad räknaren visar: `f'{x:02d}'`, om variabeln som ska strängifieras är x.

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.