Programmering og Problemløsning
Arbejdsseddel 12 - gruppe opgave

Ken Friis Larsen kflarsen@diku.dk

7. januar , 2023 – 14. januar, 2023

Afleveringsfrist: lørdag d. 14. januar, 2023, kl. 22:00.

Læringsmål

Få praktisk erfaring med at arbejde med Python, specifikt med:

- At lave små moduler og bruge dem fra andre filer
- At kunne definere klasser og instantiere objekter
- At bruge klasser, som er organiseret i et hierarki

Opgaverne er opdelt i repetition-/øve- og afleveringsopgaver. I denne periode skal I arbejde i grupper med jeres afleveringsopgaver. Regler for gruppe- og individuelle afleveringsopgaver er beskrevet i "Noter, links, software m.m." → "Generel information om opgaver".

Repetition 1 – Ugeseddel 5i

Løs opgave 5ø0, men denne gang i Python. Det vil sige, lav et vektor modul i filen `vec.py`, og lav en anden fil `using_vec.py` som importer `vec` og bruger funktionerne.

Kig gerne på jeres gamle besvarelser i F#.

Eksperimenter evt. med forskellige repræsentation af vektorer: tupler, lister, dictionaries, objekter, namedtuples.

Repetition 2 – Ugeseddel 9i og 10g

Løs opgave 9ø0, 9ø1, 9ø4, 9ø5 fra ugeseddel 9i, samt opgave 10ø0 – 10ø4 men denne gang i Python. Kig gerne på jeres gamle besvarelser i F#.

Repetition 3 – Critters 2.0

I denne opgave skal vi endnu en gang repræsentere *critters*, digitale væsener i et endnu ikke færdiggjort spil, fra ugeseddel 11i. Men denne gang ved hjælp af objekter.

Løs opgave 5 fra ugeseddel 11i, men brug objekter til at repræsentere critters.

Øvelse 1 – Figurer

Givet følgende F# kode til at arbejde med figurer:

```
type colour = string
type point = int * int
type Figure =
  | Circle of point * int * colour
  | Rectangle of point * point * colour

let contains figure (x,y) =
    match figure with
    | Circle ((cx,cy), r, col) ->
        (x-cx)*(x-cx) + (y-cy)*(y-cy) <= r*r
    | Rectangle ((x0,y0), (x1,y1), col) ->
        x0<=x && x <= x1 && y0 <= y && y <= y1

let mvPoint (x, y) dx dy = (x + dx, y + dy)
let move figure dx dy =
    match figure with
    | Circle (center, r, col) ->
        Circle (mvPoint center dx dy, r, col)
    | Rectangle (topL, botR, c) ->
        Rectangle(mvPoint topL dx dy, mvPoint botR dx dy, c)
```

Det vil sige, vi har to slags `Figure`: `Circle` og `Rectangle`; og vi har to funktioner der arbejder med figurer: `contains` der kan afgøre om en figur indeholder et punkt, og `move` der kan flytte en figur.

a. Skriv tilsvarende kode, men hvor vi bruger klasser i stedet. Det vil sige, at I skal implementere mindst to klasser: `Circle` og `Rectagle` og begge disse klasser skal have mindst to metoder: `contains` og `move`.

   "Tilsvarende kode" betyder i dette tilfælde at `move` ikke skal returnere en ny figur, men skal mutere figuren. Hvorfor kun "mindst to klasser"? Det er op til jer, om I har brug for en tredje klasse `Figure` som `Circle` og `Rectangle` nedarver fra.

   I bestemmer selv om I vil implementere jeres klasser i både F# og Python, eller kun i Python.

b. Udvid med en ekstra slags figur `Square`.

c. Udvid med en ekstra metode `area` som kan beregne arealet af en figur.

d. Lav udvidelserne b. og c. på den given funktionelle F# kode.

Aflevering 12g – Data Pipelines

I denne opgave skal I arbejde med *data pipelines* der opbygges af forskellige skridt (*steps*). Data pipelines har følgende egenskaber:

- de kan bruges til at beregne et resultat;
- de kan dokumentere sig selv

I skal bruge objekter til at modellere de forskellige skridt i en pipeline. Interfacet for et step i en pipeline er et objekt som implementerer følgende metoder (protokol):

- `apply(Input) -> Result`, dvs. en metode der tager et argument og returnerer et resultat (men det er ikke specificeret hvad typerne for `Input` og `Result` skal være).
- `description() -> str`, en metode som returnerer en streng der beskriver hvad skridtet gør.

Hvis en klasse implementerer disse to metoder, siger vi at den implementerer `Step` interfacet, eller at klassen er et `Step`.

For eksempel, så implementerer følgende klasse `Step` interfacet:

```
class DoNothing:
    """Do nothing"""
    def apply(self, inp):
        return inp
    def description(self) -> str:
        return self.__doc__.lower()
```

Bemærk at metoden `description` udnytter at klassen har en docstring (som den kalder metoden `lower()` på). Det er ikke et krav at jeres kode gør det samme, men det kan nogle gange være handy.

Opgave A

Implementer følgende klasser som `Steps`

- `AddConst` som man kan konstruere ved at give en værdi `c`. Hvor `apply` skal lægge input sammen med `c` ved hjælp af `+` og returnere resultatet.

- `Repeater` som man kan konstruere ved at give et heltal, `num`. Hvor `apply` skal returnere en liste med `num` elementer der alle er input. Hvis `num` er mindre end nul, returneres den tomme liste.

- `GeneralSum` som man kan konstruere ved at give et neutralt element og en (binær associativ) operator. Hvor `apply` antager at input er en iterable og returnerer summen beregnet via det givne neutrale element og den givne operator.

  (I F# vil det svare til `Seq.fold`, hvor operatoren er funktionen og det neutrale element er start-akkumulatoren)

Brug nedarvning fra `GeneralSum` til at definere følgende to `Steps`:

- `SumNum` skal summe med `+` og har neutralt element `0`.
- `ProductNum` skal gange med `*` og har neutralt element `1`.

Eksempel på brug:

```
>>> adderOne = AddConst(1)
>>> adderOne.apply(41)
42
>>> quintiplyer = Repeater(5)
>>> quintiplyer.apply(42)
[42, 42, 42, 42, 42]
>>> summer = SumNum()
>>> summer.apply([1,2,3,4,5,6,7,8,9,10])
55
>>> proder = ProductNum()
>>> proder.apply([1,2,3,4,5])
120
```

Opgave B

Implementer klassen `Map` som et `Step`. `Map` konstrueres ved at give et *andet* `Step`. Hvor `apply` skal kalde `apply` på det indre `step`, for alle elementer i input og returnere resultatet. Kræver at input er iterable.

Opgave C

Implementer klassen `Pipeline` som et `Step`. `Pipeline` konstrueres ved at give en liste af `Step`. Hvor `apply` skal kalde `apply` på de indre `Step` i rækkefølge, hvor resultatet af et `Step` bliver givet som argument til det efterfølgende `Step`, og tilsidst returnere resultatet af det sidste `Step`.

Udover metoderne `apply` og `description`, så skal `Pipeline` også have metoden `add_step`, som tilføjer et `Step` til pipelinen.

Eksempel på brug af `Pipeline`:

```
>>> pipeline = Pipeline([
...     AddConst(45),
...     Repeater(3),
...     Map(AddConst(-3)),
...     DoNothing(),
...     ProductNum()])
>>>
>>> pipeline.apply(0)
74088
>>> pipeline.description()
'[add 45 -> repeat 3 times, as a list -> add -3 to each
        element -> do nothing -> multiply all elements]'
```

Bemærkning: I har en vis frihed med hensyn til formatet af resultatet fra `description()`. Ovenstående er et forslag til, hvordan det evt kan se ud. Dette gælder for alle `Steps` ikke kun `Pipeline`.

Frivillig udvidelse: Et problem med lange pipelines kan være at debugge hvor det er gået galt, hvis et skridt fejler. Sørg derfor for at `Pipeline.apply` kan håndtere at et indre `Step` fejler, og kan give noget ekstra information om hvad der er gået godt og hvad der er gået galt. Det kan fx se ud som følger:

```
>>> faulty = Pipeline([
...     AddConst(45),
...     Repeater(3),
...     AddConst(-3),
...     DoNothing(),
...     ProductNum()])
>>>
>>> faulty.apply(0)
Traceback (most recent call last):
  ...
  ...
Exception: Error 'can only concatenate list (not "int") to
list' at step 3 'add -3',
after succesful steps:
  add 45
  repeat 3 times, as a list
```

Opgave D

I denne delopgave skal I benytte jeres *data pipelines* til at beregne statistik over *critters*.

Implementer klassen `CsvReader` som et `Step`. `apply` skal tage imod et filnavn, indlæse filen vha. `csv`-modulet og returnere en liste af dictionaries.

Implementer klassen `CritterStats` som et `Step`. `apply` skal tage imod en liste af dictionaries (*critter data*), tælle hvor mange forekomster der er af hver farve i datasættet og returnere en dictionary hvor farven er *key* og antal critters med den farve er *value*.

Implementer klassen `ShowAsciiBarchart` som et step. `apply` skal printe en tabel på formatet `farve: ***`, hvor antallet af `*` svarer til antallet af critters med den farve, og returnere inputtet uden at modificere det. Hvis der er 2 critters med farven `'red'` og 5 med farven `'violet'` skal output være:

```
red   : **
violet: *****
```

Dvs, kolon i tabellen bør stå lodret over hinanden.

Opgave E

Lav (mindst) to steps I selv finder på. Her er nogle forslag til idéer:

- `Square` eller `Cube` der svarer til henholdsvis `x*x` og `x*x*x`
- `Average` der udregner gennemsnit af input
- `StandardDeviation` der udregner standardafvigelsen af input
- `Stringify` der konverterer input til en streng-repræsentation
- `Sort` der sorterer input
- `FunStep` der applicerer en arbitrær given funktion på input
- `ShowMatplotlibBarchart` som viser et bar-chart vha `matplotlib`

Opgave F

Test jeres forskellige `Steps` fra de andre opgaver. Jeres tests skal kunne køres fra en separat fil kaldet `testPipeline.py`, vi foreslår at I bruger enten `doctest` eller `unittest` bibliotekerne.

Krav til afleveringen

Afleveringen skal bestå af

- en zip-fil, der hedder `12g.zip`

Zip-filen skal indeholde:

- filen `README.txt` som er en tekstfil indeholdende jeres navn, en beskrivelse af de vigtigste beslutninger I taget i forhold til jeres kode, hvad I har fokuseret på i jeres test, samt en beskrivelse af hvordan man kører jeres test.

- en `src` mappe med mindst følgende filer: `pipeline.py` og `testPipeline.py`

Det skal være muligt at køre opgave A, B, C, D og F uden at installere ekstra pakker end dem fra Pythons standard bibliotek. Det vil sige, at det er jer frit at bruge eksterne pakker til jeres `Steps` i opgave E. Men hvis I bruger eksterne pakker, så skal de importeres i et indre scope, og ligeledes skal tests af disse `Steps` køres separat fra `testPipeline.py` (fx fra `testExtraPipeline.py`).

God fornøjelse.