Högre ordrningens funktioner (Higher Order Functions)

I detta dokument går vi igenom några så kallade högre ordningens funktioner (higer-order functions https://en.wikipedia.org/wiki/Higher-order_function), vad de är till för och när man kan använda dem när man programmerar. Som ni kommer att märka kan man utföra samma sak på flera olika sätt och oftast är inget mer rätt än något annat.

I obligatoriska uppgiften 4 (OU4) bör du använda så många av följande koncept och funktioner som möjligt. Observera att man ofta kan använda dem enskilt eller i olika kombinationer med varandra för att uppnå samma mål.

Högre ordningens funktioner tar antingen funktioner som argument eller returnerar funktioner. Koncepten kommer ursprungligen från så kallade funktionella språk (https://en.wikipedia.org/wiki/Functional_programming), några exempel är Haskell, Erlang, OCaml och Scheme. De flesta moderna språk har denna typ av funktionalitet. Den intresserade kan läsa mer här

https://www.guru99.com/functional-programming-tutorial.html.

Vi ska i detta dokument gå igenom några viktiga koncept, som du kan använda i Python.

1. List comprehensions (https://en.wikipedia.org/wiki/List_comprehension)
2. Lambda-funktioner (https://en.wikipedia.org/wiki/Anonymous_function)
3. `map()` (https://en.wikipedia.org/wiki/Map_(higher-order_function))
4. `functools.reduce()` (https://en.wikipedia.org/wiki/Fold_(higher-order_function))
5. `filter()` (https://en.wikipedia.org/wiki/Filter_(higher-order_function))
6. `zip()` (https://en.wikipedia.org/wiki/Convolution_(computer_science))

1 List comprehensions

När man ska konstruera lite mer komplicerade listor i Python kan det vara enklare och tydligare att använda så kallade *list comprehensions* än att använda `for`-loopar etc. List comprehensions består av ett uttryck med `for`-loopar och `if-else`-uttryck skrivet inom hakparenteser `[]`.

Ett enkelt exempel är att du vill skapa en lista med talen 1 till 4 kvadrerade. Detta kan skapas med

```
>>> lst = [ii**2 for ii in range(1,5)]
>>> lst
[1, 4, 9, 16]
```

Detta är bara ett kortfattat sätt att skriva:

```
lst = []
for ii in range(1,5):
    lst.append(ii*2)
```

Om du istället hade velat ha en lista där du tar $\Gamma(n)$ för $n = 2, 4, 6, 8$ kan du skriva

```
>>> import math
>>> [math.gamma(ii) for ii in range(1,10) if ii % 2 == 0]
[1.0, 6.0, 120.0, 5040.0]
```

Det som sker är att `for`-loopen sätter variabeln `ii` till $1, 2, \ldots, 9$, och för varje värde testar den om resten för `ii % 2` är lika med noll. Om den är det, så utförs `math.gamma(ii)`.

```
import math
lst = []
for ii in range(1,10):
    if ii % 2 == 0:
        lst.append(math.gamma(ii))
```

Ett annat exempel är följande,

```
>>> [[0 for ii in range(0,2)] for jj in range(0,3)]
[[0, 0], [0, 0], [0, 0]]
```

Här har du en inre list comprehension som skapar en lista av längd två, med nollor i. Den yttre skapar sedan en lista av längd tre med dessa listor som element.

En bra djupare genomgång, med exempel, av list comprehensions kan du t.ex. hitta på

https://www.datacamp.com/community/tutorials/python-list-comprehension

2 Lambda-funktioner

Lambda-funktioner finns i de flesta språk, och kallas även för anonyma funktioner. T.ex. i MATLAB uttrycks de som `@(x)x^2`, och om man vill integrera funktionen kan man skriva `integral(@(x)x^2,0,1)` istället för att skriva en hel "riktig" funktion.

Motsvarande i Python är att funktionen är enkel (den kan skrivas på en rad) och att man då slipper defineira en funktion med `def funktionsnamn:`. Lambda-funktioner ska främst användas för funktioner som bara används en "kort tid". (t.ex. bara på ett ställe i koden)

Här är ett exempel på användning av lambda-funktioner.

```
>>> f = lambda x : x*2
>>> f(5)
10
```

Formatet som de definieras är `lambda <argument> : <uttryck>`. I exmeplet ovan är `x` argumnetet till funktionen (som här namnges till `f`) och uttrycket är `x*2`, dvs multiplicera `x` med två.

Ett exempel med två argument som adderas är:

```
>>> f = lambda x,y : x+y
>>> f(2,3)
5
```

Fördelen med Lambda-funktioner är dock främst i kombination med andra funktioner (speciellt funktioner som `map`, `reduce`, `filter` och `zip` som disukteras nedan).

Antag t.ex. följande exempel,

```
>>> def multiply(n):
        return lambda a : a * n
>>> double = multiply(2)
>>> triple = multiply(3)
>>> double(10)
20
>>> triple(10)
30
```

Man kan alltså skapa dessa funktioner `dubblera` och `trippla` mycket enkelt genom att nyttja den Lambda-funktionen i `multiplicera`.

En djupare genomgång av lambda-funktioner i Python kan du hitta på https://realpython.com/python-lambda/

https://realpython.com/python-lambda/

3 `map()`

I Python returnerar `map` ett map-objekt som innehåller resultaten av att man applicerar en funktion på varje element i en iterable (lista, tuple, etc.). Syntaxen är `map(<funktion>,<iterable>)`; om du har tex `map(f,[1,2,3])` kommer ett map-object med `(f(1),f(2),f(3))` att returneras. Observera att man måste konvertera det returnerade map-objektet från `map` med funktionen `list()` för att få en lista, dvs `list(map(f,[1,2,3]))`.

Ta till exempel

```
>>> import math
>>> list(map(math.gamma,range(1,5)))
[1.0, 1.0, 2.0, 6.0]
```

Så returneras $\Gamma(n)$ för $n = 1, 2, 3, 4$.

Om du vill kombinera `map` med en lambda-funktion, som i detta fall kvaderar talen 1,2,3,4, kan du skriva,

```
>>> list(map(lambda x : x**2,range(1,5)))
[1, 4, 9, 16]
```

Man kan även ha flera argument till `map`, t.ex.,

```
>>> list1=[1,2,3,4]
>>> list2=[5,6,7,8]
>>> list(map(lambda x,y : x+y, list1, list2))
[6, 8, 10, 12]
```

Ett exempel med strängar

```
>>> list_of_strings=['A','short','sentence']
>>> list(map(list, list_of_strings))
[['A'], ['s', 'h', 'o', 'r', 't'], ['s', 'e', 'n', 't', 'e', 'n', 'c', 'e']]
```

och slutgiltigen konvertering av en tuple av tuples till en tuple av listor

```
>>> a_tuple_of_tuples=((0,1),(1,2,3),(1,0))
>>> tuple(map(list, a_tuple_of_tuples))
([0, 1], [1, 2, 3], [1, 0])
```

En djupare genomgång av `map` kan du t.ex. hitta på https://realpython.com/python-map-function/.

https://realpython.com/python-map-function/

4 `functools.reduce()`

Reduce-funktionen `functools.reduce()` brukar ofta användas i kombination med `map()`; kombinationen kallas ofta MapReduce, https://en.wikipedia.org/wiki/MapReduce och är ett viktigt koncept inom Data Science of Big Data.

Syftet med `functools.reduce()` är att reducera ner en iterable (lista, tuple, etc) till ett värde. I följande exempel vill vi summera alla värden i en lista, och funktionen för detta är en lambda-funktion som definierar addition,

```
>>> import functools
>>> lst=[1,-2,3,4]
>>> functools.reduce(lambda x,y : x+y, lst)
6
```

Om man istället vill applicera Taxi-normen $\|x\|_1 = \sum_i^n |x_i|$ för en vektor x av längd n, kan man skriva det som följande (där vektorn kallas `lista`)

```
>>> import functools
>>> lst=[1,-2,3,4]
>>> functools.reduce(lambda x,y : x+y, map(abs,lst))
10
```

Först appliceras `abs` på varje element i listan `lista`, med hjälp av `map()` funktionen. Sedan reduceras den resultatet genom addition, med `functools.reduce()` och en lambda-funktion

med addition. Notera att man ej behöver konvertera map-objectet till en lista innan man använder `functools.reduce`.

En djupare genomgång av `functools.reduce()` kan hitta på https://realpython.com/python-reduce-function/.

5 `filter()`

Funktionen `filter()` kan du använda när du kan formulera en fråga som är sann eller falsk för varje element i en iterable. Till exempel, hitta alla element i en lista som är större än 2. Det kan formuleras som en funktion som returnerar `True` om argumentet är större än 2 och `False` annars,

```
def greaterThanTwo(num):
        if(num > 2):
                return True
        else:
                return False
```

Filter-funktionen kan då anropas, på en lista `a`, som

```
>>> a=[7,1,-3,4]
>>> list(filter(greaterThanTwo, a))
[7, 4]
```

Ett alternativ är att kombinera med en lambda-funktion som returnerar `True` för alla tal större än 2 och annars `False`,

```
>>> a=[7,1,-3,4]
>>> list(filter(lambda x: x>2, a))
[7, 4]
```

Lambda-funktionen `lambda x: x>2` returnerar `True` om `x>2`, annars `False`.

Ett tredje sätt för att göra samma sak är att använda en list comprehension,

```
>>> a=[7,1,-3,4]
>>> [e for e in a if e>2]   # e is shorthand for "element"
[7, 4]
```

Vad som är "bäst" kan diskuteras: list comprehension eller `lambda` + `filter`, och det är en smaksak vad man tycker är tydligast och enklast att läsa. List comprehension kan vara snabbare då den inte introducerar nån funtion för att kontrollera om `x>2`.

Vill vi hitta alla vokaler i en lista kan man använda en funktion

```
def filter_vowels(letter):
        vowels = ['a', 'o', 'u', 'å', 'e', 'i', 'y', 'ä', 'ö']
        return True if letter in vowels else False
        # note: 'return letter in vowels' would work too.
```

och anropa

```
>>> word = 'xylofon'
>>> list(filter(filter_vowels,word))
['y', 'o', 'o']
```

Lite mer exempel på hur man kan använda `filter` hittar ni t.ex. på

https://www.digitalocean.com/community/tutorials/how-to-use-the-python-filter-function

6 `zip()`

Funktionen `zip` används för att skapa en iterable av tupler av en eller flera iterables. Vi börjar med ett enkelt exempel, med en lista `djur` med tre element. Om vi gör följande

```
>>> djur = ['hund', 'katt', 'orm']
>>> list(zip(range(3),djur))
[(0, 'hund'), (1, 'katt'), (2, 'orm')]
```

skapas en lista av tre tuples med 2 element i varje, ett element från `range(3)` och ett element från `djur`.

Man kan t.ex. använda `zip` när man vill hitta index för element som har en viss egenskap, t.ex. vad är index för alla element i en lista som är större än noll?

```
>>> tal = [2, -1, 7, 9]
>>> zip_tal=list(zip(range(len(tal)),tal))
>>> [ii[0] for ii in zip_tal if ii[1]>0]
[0, 2, 3]
```

Här blir `zip_tal` listan `[(0, 2), (1, -1), (2, 7), (3, 9)]`, och sen andvänds en list comprehension för att skapa en lista av första elementet i varje tuple (`ii[0]`) om andra elementet är större än noll (`ii[1]>0`). Resultat är att elementen `[0, 2, 3]` i listan `tal` är större än noll.

Detta kan även uppnås med hjälp av en list comprehension,

```
>>> tal = [2, -1, 7, 9]
>>> [i for i in range(len(tal)) if tal[i]>0]
[0, 2, 3]

# alternatively:
>>> tal = [2, -1, 7, 9]
>>> [tal.index(e) for e in tal if e>0]
[0, 2, 3]
```

Man kan även använda `zip` med mer än två argument, t.ex.

```
>>> x = [0.1, 0.6, 0.5]
>>> y = [0.2, -0.2, 0.9]
>>> z = [0.4, 0.4, -0.1]
>>> list(zip(x,y,z))
[(0.1, 0.2, 0.4), (0.6, -0.2, 0.4), (0.5, 0.9, -0.1)]
```

Ytterligare exempel på hur man kan använda `zip` hittar du t.ex. på

https://realpython.com/python-zip-function/

Parallellprogrammering och multithreading i Python

En *process* inom programmering är ett program som körs. Datan som processen använder laddas in i datorns internminne, och programmet exekveras av datorns processor. Den utför alla beräkningar och lagrar och läser data i datorns internminne och hårddisk. Varje process har sitt eget allokerade minne.

En *tråd* (eng. thread) är en beräkningsenhet i en given process, och en process kan även köras på flera trådar samtidigt. Trådarna har då en del av processens minne, och delar det med dess andra trådar. Så kallad *multithreading* ämnar att öka effektiviteten av användningen av processorns kärnor genom att låta flera trådar skapas av en process och tilldelas olika uppgifter.

Multithreading är mindre användbart när större beräkningar ska utföras, och brukar istället användas mer för processer som involverar operativsystemet. Det handlar oftast om processer som läser från eller skriver till datorns hårddisk, eller som skickar eller tar emot data via datorns nätverk. Det är då vanligt att program sitter väntandes på nästa instruktion utan att göra något under tiden. Då går det att skynda på arbetet genom att låta ett program utföra andra uppgifter i väntan på något. Exempelvis som att utföra en beräkning i väntan på att en nedladdning ska bli klar, istället för att vänta med beräkningen tills efter att nedladdningen är klar.

Parallellprogrammering är bättre för program som enbart förlitar på processorn istället. Det rör sig oftast om program som utför stora beräkningar. Sådana program fördelas mellan processorns kärnor så att den kan utföra fler beräkningar samtidigt.

Parallellprogrammering

Vi kommer att implementera en enkel metod som härmar en beräkningsintensiv funktion. Vi kommer att köra programmet flera gånger och mäta exekveringstiden för programmet. Börja med att importera följande metoder i ett tomt Python-script `example.py`,

```
1 from time import perf_counter as pc
2 from time import sleep as pause
```

Vår metod kommer att använda `time`-modulens `sleep()`-method för att likna en process som tar en viss tid att slutföra.

```
1 def runner():
2         print("Performing a costly function")
3         pause(1)
4         print("Function complete")
```

Vi kör metoden och tar tid före och efter processen har körts. Vi skriver sedan ut hur lång tid körningen tog som en så kallad f-sträng, som helt enkelt är en enklare version av `"".format()` där vi kan sätta in variablerna direkt in i strängen.

```
if __name__ == "__main__":
        start = pc()
        runner()
        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing a costly function
Function complete
Process took 1.0 seconds
```

Vi kör nu metoden tio gånger för att se om det tar ungefär tio sekunder att köra den som förväntat.

```
if __name__ == "__main__":
        start = pc()
        for _ in range(10):
                runner()
        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Performing a costly function
Function complete
Process took 10.02 seconds
```

Då metoden anropades och exekverades tio gånger sekventiellt tog det ungefär 10.02 sekunder att slutföra programmet, vilket verkar stämma väl överens med våra förväntningar.

Genom att använda Pythons `multiprocessing`-modul kan vi köra metoden flera gånger parallellt. Dagens datorer är oftast flerkärniga, och kan därför köra flera program samtidigt. Den här modulen kommer att hjälpa oss med att skicka arbetsuppgifter till de olika kärnorna för att köra delar av programmet parallellt. Importera modulen längst upp i din kod.

```
import multiprocessing as mp
```

För att skapa en process måste vi skapa ett processobjekt med metoden som vi vill köra som parametern `target`. (Det finns även en parameter vid namn `args` som kan användas för att skicka in en lista till metoden.) Vi skapar nu flera processobjekt genom att skriva följande kod.

```
p1 = mp.Process(target=runner)
p2 = mp.Process(target=runner)
```

För att starta processerna behöver vi anropa metoden `start` som är associerad med processens objekt.

```
p1.start()
p2.start()
```

Låt oss nu beräkna hur länge programmet kommer att köras.

```
if __name__ == "__main__":
        start = pc()
        p1 = mp.Process(target=runner)
        p2 = mp.Process(target=runner)
        p1.start()
        p2.start()
        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Process took 0.02 seconds
Performing a costly function
Performing a costly function
Function complete
Function complete
```

Det som vi ser nu är att processerna verkar ha tagit 0.02 sekunder. Det är så klart fel då att vi kan se att det finns kvar utskrifter efter den sista utskriften i vår metod. Det som hände här var att processerna startade samtidigt, men att programmet fortsatte att köras trots att processerna inte var klara än. Problemet här är att vi inte vet exakt hur lång tid programmet tog att utföra processerna eftersom att metoderna inte längre körs sekventiellt.

Vi löser det här problemet genom att använda metoden `join` för att vänta in processerna tills dess att alla är klara.

```
p1.join()
p2.join()
```

Vi mäter nu hur lång tid programmet egentligen tog att köra.

```python
if __name__ == "__main__":
        start = pc()
        p1 = mp.Process(target=runner)
        p2 = mp.Process(target=runner)
        p1.start()
        p2.start()
        p1.join()
        p2.join()
        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing a costly function
Performing a costly function
Function complete
Function complete
Process took 1.1 seconds
```

Nu ser vi hur programmet enbart tog ungefär en sekund att köra, trots att metoden kördes två gånger och stannades en sekund varje gång. Vi halverade alltså exekveringstiden av programmet.

Om vi vill köra många processer samtidigt så kan vi starta och köra dem i en loop. Om vi vill samla in dem så måste vi däremot göra det utanför loopen. Då måste vi lagra våra processer i en lista så att vi kan iterera över listan av processer och samla in dem då de är klara. Om vi samlar in dem direkt i loopen så kommer de att köras sekventiellt igen.

Här är generisk kod för att köra 10 parallella körningar `some_method` med argument `some_var`.

```python
processes = []
for _ in range(10):
        p = mp.Process(target=some_method, args=[some_var])
        processes.append(p)
for p in processes:
        p.start()
for p in processes:
        p.join()
```

Futures

Om vi även vill få tillbaka resultaten av våra metoder så måste vi använda klassen `Manager` för att använda oss av några arbetsmetoder som lagrar värden åt oss. Den här klassen kan vara lite svårhanterlig, så vi kommer istället att betrakta en annan Python-modul som kan köra parallellprogram och hantera insamlingen och returerna åt oss, som på så sätt är mer lätthanterlig. Den här modulen är bara en av många sätt att visa hur de här processerna fungerar innan vi använder den mer användarvänliga modulen `concurrent.futures`.

Ta bort importen av `multiprocessing`, och importera istället `concurrent.futures`.

```python
import concurrent.futures as future
```

Modulen Futures har en gruppexekverare som är lättast att köra i en så kallad context manager (ett `with`-statement i Python). Vi kallar då på metoden `submit` genom att skicka in metoden som vi vill köra tillsammans med eventuella inparametrar som vi vill ska följa med vår metod då den körs. För att sedan komma åt våra returvärden kallar vi på metoden `result` för att komma åt processobjekten. Den här modulen låter oss samla processerna genom att automatiskt vänta på processer tills dess att de är klara innan vi lämnar `with`-delen av koden.

```
with future.ProcessPoolExecutor() as ex:
        p1 = ex.submit(some_method, some_arg, some_other_arg, ...) # Starts first
        ↪  process
        p2 = ex.submit(some_method, some_arg, some_other_arg, ...) # Starts second
        ↪  process

        r1 = p1.result() # Program waits until p1 is complete before assigning r2
        r2 = p2.result()

print("all done") # Will be printed once all processes are completed
```

Vi börjar med att uppdatera vår metod `runner` genom att ge den en inparameter och en returnsträng.

```
def runner(n):
        print(f"Performing costly function {n}")
        pause(n)
        return f"Function {n} has completed"
```

Istället för att köra processerna i en loop kan vi använda exekveringsmetodens metod `map`. Med *exekveringsmetod* menar vi den metod som exekverar metoden `map`. Python har en inbyggd metod map som tar en metod och en lista av variabler som sedan sedan itereras över för att exekvera metoden varje element i listan. Användningen av den inbyggda metoden visas i följande exempel.

```
>>> def sq(x):
        return x**2

>>> for x in map(sq, [2,3,4,5]):
        print(x)
4
9
16
25
```

Låt oss nu deklarera en lista heltal som vi tar som inparameterar till vår metod `runner`. Vi skapar sedan en variabel vid namn `results` för att lagra våra returvärden som exekveringsmetodens metod `map` returnerar. Vi skickar sedan vår metod tillsammans med listan av heltal in i anropet. Slutligen så itererar vi över det returnerade objektet `map` och skriver ut resultaten som det returnerar.

```
if __name__ == "__main__":
        start = pc()

        with future.ProcessPoolExecutor() as ex:
                p = [5, 4, 3, 2, 1]
                results = ex.map(runner, p)

                for r in results:
                        print(r)

        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing costly function 5
Performing costly function 4
Performing costly function 3
Performing costly function 2
Performing costly function 1
Function 3 has completed
Function 2 has completed
Function 1 has completed
Function 5 has completed
Function 4 has completed
Process took 5.21 seconds
```

Som vi ser så exekverades metoderna i en annan ordning än i den som de anropades. Det sker eftersom att det inte finns något sätt att veta hur upptagen en given processorkärna är då programmet skickar uppgiften till den. Så metoderna kommer att startas i olika ordning varje gång du kör programmet. Den process som kör processerna håller däremot koll på deras returvärden och försäkrar oss om att de returneras i den ordning som de exekverades i. Modulen `futures` har även en metod vid namn `as_completed` som returnerar resultaten i den ordning som de kom in. (Den kan inte användas tillsammans med `map`, men det går att använda den tillsammans med metoden submit som tidigare användes i en loop).

Multithreading

Användningen av modulen `concurrent.futures` genom att köra flera trådar är inte svårare än att ändra `ProcessPoolExecutor` i en klass till `ThreadPoolExecutor`. I övrigt är proceduren precis densamma.

```
if __name__ == "__main__":
        start = pc()

        with future.ThreadPoolExecutor() as ex:
                p = [5, 4, 3, 2, 1]
                results = ex.map(runner, p)

                for r in results:
                        print(r)

        end = pc()
        print(f"Process took {round(end-start, 2)} seconds")
```

```
$ python3 example.py
Performing costly function 5
Performing costly function 4
Performing costly function 3
Performing costly function 2
Performing costly function 1
Function 5 has completed
Function 4 has completed
Function 3 has completed
Function 2 has completed
Function 1 has completed
Process took 5.01 seconds
```

Som ni ser så är utdatan från programmet lite annorlunda. Metoderna exekveras här i den ordning som de anropades. Det beror på att vi inte längre använder flera kärnor för att utföra uppgiften. Syftet med den här koden är så klart bara att illustrera proceduren, men det är inte beräkningsintensivt eller knutet till operativsystemet, så trådproceduren fungerar precis lika bra i det här fallet, och processerna slutfördes återigen efter cirka fem sekunder.

Länkar till den som vill lära sig mer, och se andra exempel:

https://www.machinelearningplus.com/python/parallel-processing-python/

https://www.geeksforgeeks.org/parallel-processing-in-python/