

Højere ordens funktioner

Indtil nu har vi set på funktioner som en måde at strukturere vores kode på. Funktioner kan tage input parametre og returnere output værdier.

I det følgende skal vi se på funktioner som "førsteklasses borgere" i Python. Det betyder at funktioner kan behandles som enhver anden datatype i Python. Vi kan tildele funktioner til variabler, vi kan sende funktioner som argumenter til andre funktioner, og vi kan returnere funktioner fra andre funktioner.

Vi kalder også disse funktioner for højere ordens funktioner (higher-order functions).

Funktioner som argumenter

En højere ordens funktion er en funktion, der enten tager en eller flere funktioner som argumenter, eller returnerer en funktion som resultat.

Lad os se på et eksempel på en højere ordens funktion, der tager en funktion som argument:

```
def anvend_funktion(funktion, værdi):  
    return funktion(værdi)  
def kvadrat(x):  
    return x * x  
resultat = anvend_funktion(kvadrat, 5)  
print(resultat) # 25
```

I dette eksempel definerer vi en funktion `anvend_funktion`, der tager en funktion `funktion` og en værdi `værdi` som argumenter. Den anvender derefter funktionen `funktion` på `værdi` og returnerer resultatet. Vi definerer også en simpel funktion `kvadrat`, der returnerer kvadratet af et tal. Når vi kalder `anvend_funktion` med `kvadrat` og `5`, får vi resultatet `25`.

Endnu et eksempel kunne være en funktion, der returnerer en anden funktion:

```
def lav_potensfunktion(n):  
    def potensfunktion(x):  
        return x ** n  
    return potensfunktion  
kvadratfunktion = lav_potensfunktion(2)  
kubikfunktion = lav_potensfunktion(3)  
print(kvadratfunktion(4)) # 16  
print(kubikfunktion(4))   # 64
```

I dette eksempel definerer vi en funktion `lav_potensfunktion`, der tager et tal `n` som argument og returnerer en ny funktion `potensfunktion`, der hæver et tal `x` til potensen `n`. Vi kan derefter oprette specifikke potensfunktioner som `kvadratfunktion` og `kubikfunktion` ved at kalde `lav_potensfunktion` med henholdsvis `2` og `3`.

Eksemplerne ovenfor viser, hvordan vi kan sende en funktion som et argument til en anden funktion, hvilket er en vigtig egenskab ved højere ordens funktioner. Det adskiller sig fra almindelige funktioner, der kun tager værdier som argumenter.

Lambda udtryk og anonym funktioner

Ret ofte når vi arbejder med højere ordens funktioner ønsker vi ikke at definere en hel funktion med et navn, men i stedet bare bruge en lille funktion et enkelt sted i koden. Her kan vi bruge lambda udtryk til at definere anonyme funktioner. Anonyme funktioner er funktioner uden et navn, som vi kan definere "on the fly" og bruge med det samme.

Lad os se på et eksempel:

```
def anvend_funktion(funktion, værdi):  
    return funktion(værdi)  
resultat = anvend_funktion(lambda x: x * x, 5)  
print(resultat) # 25
```

I dette eksempel bruger vi et lambda udtryk `lambda x: x * x` som argument til funktionen `anvend_funktion`. Dette lambda udtryk definerer en anonym funktion, der returnerer kvadratet af `x`. Når vi kalder `anvend_funktion` med dette lambda udtryk og `5`, får vi resultatet `25`.

I et andet eksempel kan vi bruge lambda udtryk til at undersøge om et tal er lige eller ulige:

```
tal_liste = [1, 2, 3, 4, 5, 6]  
lige_tal = list(filter(lambda x: x % 2 == 0, tal_liste))  
print(lige_tal) # [2, 4, 6]
```

I dette eksempel bruger vi `filter` funktionen sammen med et lambda udtryk `lambda x: x % 2 == 0` for at filtrere listen `tal_liste`, så kun de lige tal bliver inkluderet i resultatet.

Mere generelt har lambda udtryk følgende syntaks:

```
lambda argumenter: udtryk
```

Hvor `argumenter` er de input parametre, som funktionen tager, og `udtryk` er den enkelt linje kode, der returnerer en værdi.

Herunder nogle vidt forskellige eksempler på lambda udtryk:

```
# Lambda udtryk til at lægge to tal sammen  
sum_funktion = lambda x, y: x + y  
print(sum_funktion(3, 5)) # 8  
# Lambda udtryk til at finde maksimum af to tal  
maks_funktion = lambda x, y: x if x > y else y
```

```
print(maks_funktion(10, 7)) # 10
# Lambda udtryk til at beregne længden af en streng
længde_funktion = lambda s: len(s)
print(længde_funktion("Hej verden")) # 11
```

Øvelse: Anonyme funktioner med lambda udtryk

Lav en funktion `anvend_funktion`, der tager en funktion og en værdi som argumenter, og anvender funktionen på værdien. Brug derefter et lambda udtryk til at beregne kubikværdien af et tal ved at kalde `anvend_funktion` med lambda udtrykket og et tal efter eget valg.

Øvelse: Lambda udtryk

Forklar hvad følgende lambda udtryk gør, og hvad resultatet bliver når det kaldes med det angivne argument:

```
# Lambda udtryk 1:
resultat = (lambda x: x ** 3 + 2 * x)(4)
print(resultat)

# Lambda udtryk 2:
resultat = (lambda s: s.upper() + "!!!")("hej")
print(resultat)

# Lambda udtryk 3:
resultat = (lambda a, b: a if a > b else b)(10, 20)
print(resultat)

# Lambda udtryk 4:
resultat = (lambda x: x[::-1])("Python")
print(resultat)
```

Indbyggede højere ordens funktioner

Python har flere indbyggede højere ordens funktioner, såsom `map`, `filter`, `zip`, `sorted`, `min`, `max`, `any` og `all`, som vi kan bruge til at arbejde med lister og andre iterable objekter på en mere funktionel måde.

I det følgende skal vi se nærmere på nogle af disse funktioner.

map funktion

Funktionen `map` anvender en given funktion på hver enkelt element i en iterable (som en liste) og returnerer et map-objekt (som kan konverteres til en liste).

Et eksempel:

```
def ermyndig(alder):
    return alder >= 18
```

```
aldre = [15, 22, 18, 30, 16]
myndighedsstatus = list(map(ermyndig, aldre))
print(myndighedsstatus) # [False, True, True, True, False]
```

I dette eksempel definerer vi en funktion `ermyndig`, der returnerer `True`, hvis en given alder er 18 eller ældre, og `False` ellers. Vi bruger derefter `map` til at anvende denne funktion på hver alder i listen `aldre`, og konverterer resultatet til en liste, hvilket er `list` som konverterer map-objektet til en liste.

Endnu et eksempel med at udregne vægtet karaktergennemsnit:

```
karakterer = [10, 12, 7, 4, 2]
vægte = [2, 3, 1, 1, 1]
vægtede_karakterer = list(map(lambda k, v: k * v, karakterer, vægte))
gennemsnit = sum(vægtede_karakterer) / sum(vægte)
print(gennemsnit) # 9.0
```

I dette eksempel bruger vi `map` sammen med en `lambda` funktion til at beregne de vægtede karakterer ved at multiplicere hver karakter med dens tilsvarende vægt. Vi summerer derefter de vægtede karakterer og dividerer med summen af vægtene for at få det vægtede gennemsnit.

filter funktion

Funktionen `filter` anvender en given funktion på hver enkelt element i en iterable og returnerer et filter-objekt, der indeholder de elementer, hvor funktionen returnerer `True`.

Et eksempel fra en gymnasiefest hvor vi kun vil have folk med billet ind:

```
def har_billet(navn):
    billetliste = ['Anna', 'Bo', 'Carla', 'David']
    return navn in billetliste
deltagere = ['Anna', 'Erik', 'Bo', 'Freja', 'Carla', 'Gustav']
tilladte_deltagere = list(filter(har_billet, deltagere))
print(tilladte_deltagere) # ['Anna', 'Bo', 'Carla']
```

I dette eksempel definerer vi en funktion `har_billet`, der returnerer `True`, hvis et givet navn er på billetlisten, og `False` ellers. Vi bruger derefter `filter` til at filtrere listen `deltagere`, så kun de navne, der har billet, bliver inkluderet i resultatet.

Herudover kan vi bruge `filter` til at finde alle der er dumpet i en prøve:

```
karakterer = [10, 12, 7, 4, 2, 0, -3]
dumpepede = list(filter(lambda k: k < 2, karakterer))
print(dumpepede) # [0, -3]
```

I dette eksempel bruger vi `filter` sammen med en `lambda` funktion til at finde alle karakterer, der er mindre end 2, hvilket indikerer dumpede elever.

zip funktion

Funktionen `zip` tager to eller flere iterable objekter og kombinerer dem til en enkelt iterable af tuples, hvor hver tuple indeholder elementer fra de input iterable objekter på samme position.

Et eksempel:

```
navne = ['Anna', 'Bo', 'Carla']
aldre = [20, 22, 19]
kombineret = list(zip(navne, aldre))
print(kombineret) # [('Anna', 20), ('Bo', 22), ('Carla', 19)]
```

I dette eksempel bruger vi `zip` til at kombinere to lister, `navne` og `aldre`, til en liste af tuples, hvor hver tuple indeholder et navn og den tilsvarende alder.

Herunder et eksempel på at bruge `zip` hvor vi kalder to lister med to forskellige funktioner:

```
def kvadrat(x):
    return x * x
def kubik(x):
    return x * x * x
tal = [1, 2, 3, 4]
resultater = list(zip(map(kvadrat, tal), map(kubik, tal)))
print(resultater) # [(1, 1), (4, 8), (9, 27), (16, 64)]
```

I dette eksempel bruger vi `map` til at anvende funktionerne `kvadrat` og `kubik` på listen `tal`, og derefter bruger vi `zip` til at kombinere resultaterne til en liste af tuples, hvor hver tuple indeholder kvadratet og kubikken af hvert tal.

Øvelser

- Lav først to lister bestående af byer og deres respektive postnumre, så de til hinanden tilhørende værdier står på de samme pladser i de respektive lister. Brug `zip()` og `list()` funktionerne til at lave en sammenkædet liste af tupler fra de to lister. Print den sammenkædet liste.
- Brug `zip` og `dict` funktioner til at lave en dictionary der har key-value fra de to lister i punkt 1.
- Lav to lister af lister med lige mange lister i hver. Brug `zip` til at sammenkæde de enkelte lister i de to lister.
- Brug en for løkke, `enumerate` funktionen til at printe to ens-længdede lister der består af hhv. navne og aldre, så vi får : 0 navn alder, 1 navn alder, osv.
- Prøv at bruge `zip` på at sammenkæde mindst tre lister bestående af by, postnr og byens population.
- Overvej selv et tilfælde hvor det kan være smart at bruge `zip`-funktionen

Sorted funktion

Funktionen `sorted` tager uformelt en liste som input og returnerer en ny liste med de samme elementer sorteret i stigende rækkefølge. Vi kan også angive en brugerdefineret sorteringsnøgle ved hjælp af en funktion eller et lambda udtryk.

Et eksempel:

```
tal = [5, 2, 9, 1, 5, 6]
sorterede_tal = sorted(tal)
print(sorterede_tal) # [1, 2, 5, 5, 6, 9]
```

I dette eksempel bruger vi `sorted` til at sortere listen `tal` i stigende rækkefølge.

Herunder et eksempel hvor vi sorterer en liste af navne efter længden af navnene:

```
navne = ['Anna', 'Bo', 'Carla', 'David', 'Eve']
sorterede_navne = sorted(navne, key=lambda navn: len(navn))
print(sorterede_navne) # ['Bo', 'Eve', 'Anna', 'David', 'Carla']
```

I dette eksempel bruger vi `sorted` sammen med et lambda udtryk som sorteringsnøgle for at sortere listen `navne` baseret på længden af hvert navn.

Et sidste eksempel hvor vi sorterer en liste af tuples baseret på det andet element i hver tuple:

```
data = [('Anna', 25), ('Bo', 20), ('Carla', 30)]
sorteret_data = sorted(data, key=lambda item: item[1])
print(sorteret_data) # [('Bo', 20), ('Anna', 25), ('Carla', 30)]
```

I dette eksempel bruger vi `sorted` sammen med et lambda udtryk som sorteringsnøgle for at sortere listen `data` baseret på det andet element i hver tuple (alderen).

Øvelse: Sortering med brugerdefineret nøgle

- Lav en liste af tuples, hvor hver tuple indeholder et navn og en alder. Brug derefter `sorted` funktionen sammen med et lambda udtryk til at sortere listen baseret på alderen i stigende rækkefølge. Print den sorterede liste.
- Lav en liste af strenge med forskellige længder. Brug `sorted` funktionen sammen med et lambda udtryk til at sortere listen baseret på længden af hver streng i faldende rækkefølge. Print den sorterede liste.
- Lav en liste af ordbøger, hvor hver ordbog indeholder oplysninger om en person (navn, alder, by). Brug `sorted` funktionen sammen med et lambda udtryk til at sortere listen baseret på byens navn i alfabetisk rækkefølge. Print den sorterede liste.

any og all funktioner

Funktionerne `any` og `all` bruges til at evaluere sandhedsværdier i eksempelvis lister eller andre iterable objekter.

Funktionen `any` returnerer `True`, hvis mindst ét element i den iterable er sandt. Hvis alle elementer er falske, returnerer den `False`.

Et eksempel:

```
værdier = [0, "", None, 5, False]
resultat = any(værdier)
print(resultat) # True
```

I dette eksempel indeholder listen `værdier` flere falske værdier som 0, tom streng og None, men da der er mindst ét sandt element (5), returnerer `any` `True`.

Her et eksempel hvor `any` bruges som en højere ordens funktion:

```
def er_positiv(x):
    return x > 0
tal = [-3, -1, 0, 2, -5]
har_positiv = any(map(er_positiv, tal))
print(har_positiv) # True
```

I dette eksempel definerer vi en funktion `er_positiv`, der returnerer `True`, hvis et tal er positivt. Vi bruger derefter `map` til at anvende denne funktion på listen `tal`, og `any` til at kontrollere, om der er mindst ét positivt tal i listen.

Funktionen `all` returnerer `True`, hvis alle elementer i den iterable er sande. Hvis mindst ét element er falsk, returnerer den `False`. Et eksempel:

```
værdier = [1, "Hej", True, 3.14]
resultat = all(værdier)
print(resultat) # True
```

I dette eksempel indeholder listen `værdier` kun sande værdier, så `all` returnerer `True`. Her et eksempel hvor `all` bruges som en højere ordens funktion:

```
def er_positiv(x):
    return x > 0
tal = [3, 1, 2, 5]
alle_positive = all(map(er_positiv, tal))
print(alle_positive) # True
```

I dette eksempel bruger vi den samme `er_positiv` funktion, men denne gang kontrollerer vi med `all`, om alle tal i listen `tal` er positive. Da alle tal er positive, returnerer `all` `True`.

Øvelse: any og all funktioner

- Lav en liste af tal og brug `any` funktionen sammen med en lambda funktion til at tjekke om der er mindst ét negativt tal i listen. Print resultatet.
- Lav en liste af strenge og brug `all` funktionen sammen med en lambda funktion til at tjekke om alle strenge i listen har mere end 3 tegn. Print resultatet.
- Lav en liste af ordbøger, hvor hver ordbog indeholder oplysninger om en person (navn, alder). Brug `all` funktionen sammen med en lambda funktion til at tjekke om alle personer er over 18 år. Print resultatet.

Reduce funktion

Funktionen `reduce` er en del af `functools` modulet i Python og bruges til at anvende en funktion kumulativt på elementerne i en iterable, hvilket resulterer i en enkelt værdi. Den tager to argumenter: en funktion og en iterable.

Et eksempel:

```
from functools import reduce
def læg_sammen(x, y):
    return x + y
tal = [1, 2, 3, 4]
sum = reduce(læg_sammen, tal)
print(sum) # 10
```

I dette eksempel definerer vi en funktion `læg_sammen`, der returnerer summen af to tal. Vi bruger derefter `reduce` til at anvende denne funktion kumulativt på listen `tal`, hvilket resulterer i summen af alle tal i listen.

Her et eksempel hvor vi bruger `reduce` sammen med et lambda udtryk til at finde produktet af alle tal i en liste:

```
from functools import reduce
tal = [1, 2, 3, 4]
produkt = reduce(lambda x, y: x * y, tal)
print(produkt) # 24
```

Herunder finder vi den elev med den højeste karakter i en liste af ordbøger:

```
from functools import reduce
elever = [
    {'navn': 'Anna', 'karakter': 10},
    {'navn': 'Bo', 'karakter': 12},
    {'navn': 'Carla', 'karakter': 7}
```

```
]
bedste_elev = reduce(lambda x, y: x if x['karakter'] > y['karakter'] else y,
                     elever)
print(bedste_elev)  # {'navn': 'Bo', 'karakter': 12}
```

I dette eksempel bruger vi **reduce** sammen med et lambda udtryk til at finde den elev med den højeste karakter i listen **elever**. Lambda udtrykket sammenligner to elever ad gangen og returnerer den elev med den højeste karakter.

Øvelse: Reduce funktion

- Lav en liste af tal og brug **reduce** funktionen sammen med en lambda funktion til at finde summen af alle tal i listen. Print resultatet.
- Lav en liste af tal og brug **reduce** funktionen sammen med en lambda funktion til at finde produktet af alle tal i listen. Print resultatet.
- Lav en liste af ordbøger, hvor hver ordbog indeholder oplysninger om en person (navn, alder). Brug **reduce** funktionen sammen med en lambda funktion til at finde den ældste person i listen. Print resultatet.

Parallelisering og højere ordens funktioner

Højere ordens funktioner kan også bruges i forbindelse med parallelisering af opgaver for at udnytte flere processorkerner og forbedre ydeevnen ved behandling af store datamængder. Det er blevet aktuelt i de senere år, da datamængderne er vokset betydeligt, og der er behov for at udføre beregninger hurtigere samt udnytte moderne flerkernede processorer.

I det følgende vil vi se på MapReduce paradigmet, som er en populær tilgang til parallel databehandling, der anvender højere ordens funktioner.

MapReduce består af to hovedfaser: Map-fasen og Reduce-fasen.

- I Map-fasen anvendes en "map" funktion til at transformere input data i parallel på tværs af flere processorkerner. Hver kerne behandler en delmængde af dataene og producerer et sæt mellemresultater.
- I Reduce-fasen anvendes en "reduce" funktion til at aggregere eller kombinere mellemresultaterne fra Map-fasen til et endeligt resultat. Denne fase kan også udføres i parallel på tværs af flere kerner.

Ved at bruge højere ordens funktioner som **map** og **reduce** i MapReduce paradigmet kan vi effektivt udnytte parallelisering til at håndtere store datamængder og forbedre ydeevnen ved databehandling.

Vi skal dog være opmærksomme på, at ikke alle opgaver egner sig til parallelisering, og at der kan være overhead forbundet med at opdele og samle dataene. Derfor er det vigtigt at vurdere, om parallelisering er hensigtsmæssig for den specifikke opgave, vi ønsker at løse.

Generelt kan højere ordens funktioner som **map** og **reduce** være kraftfulde værktøjer til at implementere parallel databehandling i Python, når følgende kriterier er opfyldt:

- Opgaven kan opdeles i uafhængige delopgaver, der kan behandles parallelt.
- Dataene er store nok til at retfærdiggøre overhead ved parallelisering.
- Resultaterne fra delopgaverne kan kombineres effektivt i en Reduce-fase.

Ved overhead menes den ekstra tid og ressourcer, der kræves for at koordinere paralleliseringen, såsom opdeling af data, kommunikation mellem processer og samling af resultater.

Eksempler på opgaver, der kan drage fordel af parallelisering ved hjælp af højere ordens funktioner, inkluderer:

- Stordataanalyse: Behandling af store datasæt, såsom logfiler eller sensordata, hvor hver delmængde kan analyseres parallelt.
- Billedbehandling: Anvendelse af filtre eller transformationer på store mængder billeder, hvor hver billedbehandling kan udføres uafhængigt.
- Maskinlæring: Træning af modeller på store datasæt, hvor data kan opdeles i batches og behandles parallelt.

Ved at anvende højere ordens funktioner i forbindelse med parallelisering kan vi udnytte moderne hardware bedre og opnå betydelige forbedringer i ydeevnen ved behandling af store datamængder.

På kodeniveau kan vi bruge moduler som `concurrent.futures` eller `multiprocessing` i Python til at implementere parallelisering ved hjælp af højere ordens funktioner som `map` og `reduce`. Disse moduler giver os mulighed for at oprette tråde eller processer, der kan køre funktioner parallelt på tværs af flere kerner.

Parallellisering af MapReduce i Python

I det følgende vil vi se et eksempel på, hvordan vi kan implementere en simpel MapReduce-opgave i Python ved hjælp af `multiprocessing` modulet til at parallelisere behandlingen af data.

Målet er at tælle antallet af primtal i et stort interval ved hjælp af MapReduce paradigmet. Primtal er tal større end 1, der kun er delelige med 1 og sig selv (f.eks. 2, 3, 5, 7, 11, osv.). Primtal er vigtige i mange områder af datalogi og matematik, herunder kryptografi og talteori.

Vi vil opdele opgaven i to faser: Map-fasen, hvor vi tæller primtal i delintervaller, og Reduce-fasen, hvor vi summerer resultaterne fra Map-fasen for at få det samlede antal primtal i hele intervallet.

Til at sammenligne ydeevnen vil vi implementere både en sekventiel version og en parallel version af MapReduce-opgaven.

Først viser vi den fulde kode, og derefter forklarer vi de enkelte dele:

```
import time
from multiprocessing import Pool
from functools import reduce
```

Først importerer vi de nødvendige moduler: `time` til at måle eksekveringstid, `Pool` fra `multiprocessing` til at oprette en pulje af processer til parallel behandling, og `reduce` fra `functools` til at aggregere resultaterne i Reduce-fasen.

```
def is_prime(n):
    """Tjek om et tal er primtal (CPU-intensiv)"""
    if n < 2:
```

```

        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return False
    return True

def count_primes_in_range(start_end):
    """MAP: tæl primtal i en range"""
    start, end = start_end
    count = 0
    for n in range(start, end):
        if is_prime(n):
            count += 1
    return count

```

Ovenfor har vi to funktioner: `is_prime`, der tjekker om et tal er et primtal, og `count_primes_in_range`, der tæller antallet af primtal i et givet interval. `count_primes_in_range` fungerer som vores Map-funktion.

Endelig kommer hoveddelen af programmet, hvor vi implementerer både den sekventielle og den parallelle version af MapReduce-opgaven samt måler deres ydeevne:

```

def main():
    # Stor opgave: tæl primtal fra 0 til 500000
    total_range = 50*10**5
    # set to cpu_count() manually for progress tracking
    num_processes = 8

    # Del ranges op
    range_size = total_range // num_processes
    ranges = [(i * range_size, (i + 1) * range_size) for i in
range(num_processes)]
    ranges[-1] = (ranges[-1][0], total_range)

    # SEKVENTIEL VERSION
    print("=== SEQUENTIAL VERSION ===")
    start_tid = time.time()
    sekventiel_resultat = 0
    for i, r in enumerate(ranges):
        result = count_primes_in_range(r)
        sekventiel_resultat += result
        elapsed = time.time() - start_tid
        print(f"  Range {i+1}/{len(ranges)} færdig - Tid: {elapsed:.2f}s")
    sekventiel_tid = time.time() - start_tid
    print(f"Primtal fundet: {sekventiel_resultat}")
    print(f"Tid: {sekventiel_tid:.4f} sekunder\n")

    # PARALLEL VERSION - MAP/REDUCE

```

```

print("=== PARALLEL MAP/REDUCE VERSION ===")
start_tid = time.time()

with Pool(processes=num_processes) as pool:
    # MAP: hver proces tæller sin range
    resultater = []
    for i, result in enumerate(pool.imap_unordered(count_primes_in_range,
ranges)):
        resultater.append(result)
        elapsed = time.time() - start_tid
        print(f" Range {i+1}/{len(ranges)} færdig - Tid: {elapsed:.2f}s")
    # REDUCE: kombiner alle resultater ved at summe dem
    print(" Reducer resultater...")
    parallel_resultat = reduce(lambda x, y: x + y, resultater)
    parallel_tid = time.time() - start_tid
    print(f"Primtal fundet: {parallel_resultat}")
    print(f"Tid: {parallel_tid:.4f} sekunder")
    print(f"Speedup: {sekventiel_tid / parallel_tid:.2f}x hurtigere!")

if __name__ == "__main__":
    main()

```

I `main` funktionen definerer vi det samlede interval for primtalstælling og opdeler det i delintervaller baseret på antallet af processer.

Vi implementerer først den sekventielle version, hvor vi tæller primtal i hvert interval én ad gangen og måler tiden for hver færdiggjort interval.

Derefter implementerer vi den parallelle version ved hjælp af en pulje af processer. Vi bruger `pool.imap_unordered` til at anvende `count_primes_in_range` funktionen på hvert interval parallelt. Resultaterne samles i en liste, som vi derefter reducerer ved at summere dem for at få det samlede antal primtal. Til sidst måler vi tiden for den parallelle version og beregner speedup i forhold til den sekventielle version.

Vi bemærker, at vi kører:

```

if __name__ == "__main__":
    main()

```

Det sker for at sikre, at `main` funktionen kun køres, når scriptet eksekveres direkte, hvilket er vigtigt for korrekt funktionalitet ved brug af `multiprocessing` modulet.

Den parallelle version er markant hurtigere end den sekventielle version, især for store datamængder, hvilket demonstrerer fordelene ved at bruge højere ordens funktioner i forbindelse med parallelisering.

Øvelse: Parallel MapReduce

- Implementer en parallel MapReduce-opgave i Python ved hjælp af `multiprocessing` modulet for at tælle antallet af lige tal i et stort interval (f.eks. fra 0 til 1 million). Brug højere ordens funktioner som

`map` og `reduce` i din implementering.

- Sammenlign ydeevnen af din parallelle MapReduce-implementering med en sekventiel version, og mål tiden for begge versioner. Print resultaterne og den opnåede speedup.
- Eksperimenter med forskellige antal processer i din parallelle implementering for at se, hvordan det påvirker ydeevnen. Prøv at finde det optimale antal processer for din maskine.
- Find eller konstruer en meget stor datamængde (f.eks. en stor tekstfil eller et stort talinterval) og brug din parallelle MapReduce-implementering til at behandle denne data. F.eks. kan du tælle forekomsten af bestemte ord i en stor tekstfil eller finde summen af alle tal i et stort interval.