TDP002 – Imperativ Programmering

Introduktion

Ni kommer under kursens gång att lära er grundläggande programmering och problemlösning i Python. En förutsättning för att ni ska lära er något är självklart att ni självständigt söker efter information om de olika konstruktionerna i språket som hanteras i laborationerna. Kursboken kommer säkert vara väldigt värdefull men då nya tekniker och metoder hela tiden utvecklas och förbättras är det viktigt att även hitta externa resurser ni kan ta hjälp av. Dessutom ingår självständigt sökande efter information som kan hjälpa er lösa problem som en del av kursen och utbildningen.

Det är viktigt att ni övar på de begrepp ni inte känner er säkra på innan ni börjar med inlämningsuppgifterna och att ni tar hjälp av laborationsassistenterna på den schemalagda tiden.

Laborationsassistenter kommer finnas till hands men ni kommer i första hand få hjälp med att hitta information och verktyg för att själva kunna utforska språket och hitta lösningar till era problem. Den största delen av kursen kommer ni att lära er tolka resultatet av att exekvera er kod och era program så att ni lär er tänka som programmerare och så att programmet till sist gör så som ni tänker. Felsökning är problemlösning vilket också är kärnan i mycket programmering.

Examination & Redovisning

Redovisningen kommer ske i form av en demonstration för laborationsassistenten där ni ska förklara er kod och köra igenom ett antal testfall som otvetydigt visar att den kod ni skrivit fungerar korrekt. Allt eftersom kursen fortskrider kommer ni få bättre verktyg för att genomföra dessa testfall på ett riktigt sätt.

Testfallen skrivs i en separat fil där ni tydligt kommenterar vilket fall som testas samt det förväntade resultatet.

Innehållsförteckning

Laboration 0 – Välkommen till Python

Syftet med denna laboration är att ni ska komma igång med Python som programmeringsspråk och att ni ska få bekanta er med den miljö ni senare kommer att arbeta med. Ni kommer att gå igenom grundläggande aritmetiska uttryck och hur dessa kan användas på olika sätt i Python, hur ni lagrar data och resultat från beräkningar med hjälp av variabler och hur ni sedan utnyttjar denna data i nya beräkningar. Slutligen kommer ni även att gå igenom hur ni kan återanvända programkod i form av moduler eller program och vad som skiljer dessa två åt.

Det är viktigt att ni så snart som möjligt kommer igång och installerar Python. Se till att utnyttja den schemalagda tiden då labortionsassistenterna är närvarande och tveka inte att fråga dem om ni behöver hjälp eller har några funderingar kring laborationerna i kursen.

Övningsuppgifter

Installation av Python

Troligen så kommer Python redan vara installerat i er ubuntumiljö och ni kan kontrollera detta genom att skriva `python` i ett terminalfönster. Ifall ni inte får upp Python-prompten (>>>) så måste Python installeras. Detta gör ni med hjälp av *”The Advanced Packaging Tool”*.

Skriv följande:

```
$ sudo apt-get install python
```

Python-tolken

Ni kan läsa mer om Python-tolken och att exekvera kod interaktivt i kursboken Learning Python, kapitel 3.

Python-tolken är ett användbart verktyg där ni direkt kan skriva in olika satser och uttryck för att se hur Python tolkar dem och vilket resultat som returneras. Ni startar Python-tolken genom att skriva `python` i terminalen. Ni känner igen Python-tolken genom promten >>>. När ni ser en rad i någon Python-dokumentation (inklusive detta labortionskompendie) som börjar med >>> så innebär det att resten av raden ska skrivas in i Python-tolken.

Prova följande:

```
>>> 4
>>> 5 + 3
>>> _
>>> _ - 6
```

Python-tolken skriver alltid ut resultatet av det uttryck ni skrivit på raden under. Returvärdet sparas dessutom automatiskt i variabeln _. Detta är smidigt ifall ni vill använda Python-tolken som en (riktigt kraftfull) miniräknare.

Arbeta med Siffror

Ni kan läsa mer om att arbeta med artimetik i Python i kursboken Learning Python, kapitel 4.

I Python kan ni använda er av samma aritmetiska uttryck som ni använder i matematiken. Addition, subtraktion och multiplikation beräknas på samma sätt som ni är vana sedan tidigare. Division skiljer sig däremot åt då det finns både heltalsdivision och flyttalsdivision. Ifall ni vill använda er av division enligt den matematiska definitionen måste ni se till att minst ett av talen skrivs flyttal. Ifall ni verkligen vill använda er av heltalsdivision får ni ange detta explicit genom att använda er av operatorn `//` istället för `/`.

Prova följande:

```
>>> 3 / 2
>>> 3.0 / 2.0
>>> 3.0 / 2
>>> 3.0 // 2
>>> 3 // 2
```

Samma ordningsregler som i matematiken gäller även för Python. Multiplikation och division evalueras före addition och subtraktion. Vill ni ändra dessa regler eller förtydliga er uträkning får ni använda er av parenteser för att på så vis prioritera delar av uttrycket.

Prova följande:

```
>>> 3.0 / 2.0 * 10.0
>>> 3.0 / (2.0 * 10.0)
>>> 3.0 + 2.0 / 2.0
>>> (3.0 + 2.0) / 2.0
```

Arbeta med textsträngar

Ni kan läsa mer om att arbeta med Python och textsträngar i kursboken Learning Python, kapitel 4.

Eftersom Python är ett fullskaligt programmeringsspråk och inte bara en häftig miniräknare finns det fler datatyper än tal, till exempel textsträngar. Python tillåter att man adderar och multiplicerar textsträngar med de binära infixoperatorerna + respektive *. Binär betyder tvåvärd (tar två argument) och infix betyder att operatorn placeras mellan sina argument. Resultatet av att utföra addition och multiplikation i Python beror alltså på vilken typ av argument som ges till + respektive *.

> *”En sträng inom datalogi är en mängd som består av en ordnad följd av ett bestämt antal element ur ett givet alfabet. Den vanligaste tillämpningen av strängar är teckensträngar, som består av en ordnad följd av numeriska värden som representerar enskilda tecken ur en viss teckenkodning. Teckensträngar representerar ofta text som kan läsas av slutanvändaren i ett användargränssnitt.” – Wikipedia*

Prova följande:

```
>>> ”En ” + ”samman”
>>> _ + ”sättning”
>>> ”hej ” * 5
>>> (”hej, ” * 2 + ”hemskt mycket hej, ”) * 3
```

```
>>> raw_input(”Vad heter du? ”)
>>> print(”Hej ” + _ + ”!”)
```

Vad är en variabel?

Variabler är referenser till data som programmeraren inte känner till i förväg. Detta kan till exempel vara åldern på användaren eller resultatet av en uträkning.

> *” En variabel är något som kan ändras. Inom matematiken och datavetenskapen betecknar den ett namngivet objekt som används för att representera ett okänt värde, till exempel ett reellt tal. Variabler används i öppna utsagor. De kan anses stå i motsats till konstanter som är oföränderliga, liksom till parametrar som hålls konstanta under en given process eller beräkning.” – Wikipedia*

Prova följande:

```
>>> name = raw_input(”Vad heter du? ”)
>>> age = raw_input(”Hur gammal är du? ”)
>>> print(”Hej, ” + name + ”! Du är ” + age + ” år gammal!”)
```

Det är även vanligt att använda sig av variabler för att förenkla och förtydliga ett annars väldigt komplext uttryck. Något som är väldigt viktigt att tänka på när ni programmerar med variabler är att välja *bra namn som beskriver innehållet* på variabler. När ni arbetar med uppgifter där ni till exempel behöver lagra användarens ålder bör variabler således döpas till `age`, `user_age`, `age_of_user` eller dylikt. Att döpa variabler till `temp`, `something`, `x` eller `the_variable_I_dont_know_how_to_name` är mycket dålig sed och även om det går lite snabbare att skriva så kommer ni att förlora den tid ni tjänat första gången ni måste gå igenom och felsöka er kod. Gör det därför till en vana att alltid stanna upp och fundera varför ni egentligen behöver en variabel före ni döper den.

Hur fungerar en funktion?

Ni kan läsa mer om att arbeta med Python och funktioner i kursboken Learning Python, kapitel 16.

Funktioner används för att dela in ett större problem (ofta ett program) i mindre delproblem. Ni har redan använt er av en del funktioner i tidigare övningsuppgifter.

> *”Ordet funktion syftar inom matematiken ofta på en regel som innebär att till ett invärde associeras ett utvärde, ofta beräknat med en matematisk formel. En mycket viktig egenskap hos funktioner är att de är deterministiska (det vill säga konsekventa, så att varje invärde alltid ger samma utvärde). Detta gör att funktionen kan ses som en mekanism, en maskin, som systematiskt levererar rätt utvärde så fort man stoppar in ett invärde. I tekniska sammanhang brukar invärdet kallas ’argument’ och utvärdet för ’värdet’.” – Wikipedia*

Både `print` och `raw_input` är funktioner som ingår i grundbiblioteket för Python. Tanken med en funktion är att bryta ut en del av problemet och lösa det på ett så pass generellt sätt att ni kan använda er av funktionen i flera delar av programmet. En funktion kan även acceptera ett antal argument som används av funktionen för att komma fram till ett resultat.

Moduler

Ni kan läsa mer om att arbeta med Python och moduler i kursboken Learning Python, kapitel 21 & 22.

En av Pythons stora styrkor kommer från det faktum att Python levereras med batterier inkluderade (http://docs.python.org/tutorial/stdlib.html#batteries-included). Det finns ett stort bibliotek med moduler som kan importeras till er kod. Till exempel modulen `os` som innehåller funktioner för att interagera med opertivsystemet.

För att importera en modul till Python använder ni er av kommandot `import`. `import` fungerar så att det importerar och evaluerar den fil ni anger såvida den inte redan blivit importerad tidigare. Ifall ni vill importera en modul på nytt måste ni använda er av funktionen `reload`. Notera att `reload` inte kan användas såvida modulen inte tidigare blivit importerad.

Prova följande:

```
>>> import os
>>> import os
>>> reload(os)
```

För att få mer information om vilka funktioner som finns i en modul kan ni använda er av funktionen `dir`. Vill ni veta mer om hur en modul eller funktion fungerar och få mer information om vilka parametrar ni måste ange och vilket resultat ni kan förvänta er kan ni använda er av funktionen `help`. Använd `help` och `dir` flitigt genom laborationern när ni stöter på nya koncept för att lära er mer om dessa.

Prova följande:

```
>>> dir(os)
>>> help(os.system)
```

Inlämningsuppgifter

Uppgift 0a – Program och Moduler

All kod som ni kan skriva i Python-tolken kan ni även skriva i separata filer och sedan antingen importera dem som moduler eller köra dem som program eller applikationer. Det finns däremot vissa små skillnader så som att _ inte definieras automatiskt och att resultat inte skrivs ut automatiskt (vill ni skriva ut något använder ni er av funktionen `print`). Python-kod skriver ni filer som slutar med filändelsen `py`.

För att exekvera en fil skriver ni `python <filnamn>` i terminalen. Ni kan även köra filen direkt genom att bara skriva filens namn. För att göra filen körbar i Ubuntu använder ni kommandot `chmod u+x <filnamn>`, därefter kan filen köras som `./<filnamn>`.

Ladda ner följande fil:
http://www.ida.liu.se/~TDP002/resources/Uppgift01.py

Börja med att importera filen som en modul och titta på vilka olika funktioner som finns, kör sedan filen som ett program för att se vad som händer. När ni gjort detta öppnar ni filen i er utvecklingsmiljö för att gå igenom koden, se till att funktionerna har bra namn som tydligt beskriver deras syfte och att variabler används på rätt sätt och har tydliga namn.

Besvara slutligen följande frågor:

a) Hur kan ni avgöra ifall koden exekveras som en modul eller som ett program?
b) Vad har variablen `__name__` för värde när ni kör koden som en modul?
c) Vad har variabeln `__name__` för värde när ni kör koden som ett program?
d) Hur kan ni dokumentera funktionaliteten hos en funktion?

Laboration 1 – Datatyper och kontrollstrukturer

Syftet med denna laboration är att ni ska lära er arbeta med de vanligaste datatyperna i Python samt hur ni gör för att snabbt och enkelt arbeta er igenom listor och talföljder utan att veta resultatet i förväg. Ni kommer även lära er hur Python hanterar olika typer av villkor, i vilken ordning och till vilken grad de utvärderas och vad detta har för betydelse för er kod.

Övningsuppgifter

Strängar

Ni kan läsa mer om strängar och hur ni manipulerar dem i kursboken Learning Python, kapitel 7.

Det finns en mängd olika sätt att skriva strängar på i Python. I vanliga fall används enkla citationstecken men ni kan även använda er av dubbla citationstecken då det passar bättre. Ifall ni vill ha en sträng som innehåller båda typer av citationstecken kan ni skriva en sträng som börjar (och slutar) med tre citationstecken av någon av typerna. Sådana strängar tillåts även innehålla radbrytningar.

Prova följande:

```
>>> 'A simple string'
>>> "You'd want to use this to write in English."
>>> """I'll be able to use a lot more "special" characters within
... this string!"""
```

När ni skriver in en sträng med radbrytningar i Python-tolken märker ni att radbrytningarna representeras med `\n`. Sådana tecken kallas för escape-tecken och kan även användas i normala strängar. Detta innebär att ni måste placera ett escape-tecken före ett backslash ifall ni vill skriva ut ett backslash i er sträng. Detta kan lätt bli jobbigt och förvirrande och därför finns det något som kallas för råa strängar. Råa strängar skrivs med ett `r` framför strängen och behandlar då inte några escape-tecken.

Prova följande:

```
>>> "Jämför det här: \n ... \\"
>>> r"Jämför det här: \n ... \\"
```

Det finns två olika typer av strängar i Python, `str` och `unicode`. `str` är simpla ASCII-strängar som kan innehålla en grunduppsättning av ett latinskt alfabet och ytterligare några tecken som behövs för datorarbete. Det kan däremot bli problem med att representera specialla tecken som till exempel å, ä och ö. För detta är `unicode` mer lämpat. En `unicode`-sträng skrivs med ett `u` framför strängen.

Prova följande:

```
>>> u"en unicode sträng, inte så märkvärdig"
>>> unicode("str till unicode, teckenkodning 'utf-8'.", "utf-8")
>>> unicode(raw_input("Skriv: "), "utf-8")
```

Strängar i Python är oföränderliga datatyper (*immutable*). Ni kan inte förändra innehållet i dem så ifall ni vill modifiera en sträng får ni skapa en ny med den tidigare som utgångspunkt. Det finns väldigt många metoder för detta i Python.

Prova följande:

```
>>> example = ”Python hanterar strängar väl”
>>> example[0]
>>> example[7]
>>> example[0:6]
>>> example[16:]
>>> example[:15]
>>> len(example)
>>> example.replace(’strängar’, ’allt’)
>>> dir(””)
```

Genom att ange ett index mellan hakparenteser kan ni välja ut ett speciellt tecken från strängen. Ifall ni vill välja ut en sekvens av tecken anger ni istället ett så kallat *slice index*, det vill säga ett start-index och ett slut-index separerat med ett kolon. Genom att utelämna start-index eller slut-index tolkar Python detta som *från början av strängen fram till slut-index* respektive *från start-index till slutet av strängen*.

Listor

Ni kan läsa mer om listor och olika index i kursboken Learning Python, kapitel 8.

En lista är den mest grundläggande av de sammansatta datatyperna och en av de mest använda datatyperna. Denna centrala position framhävs i Python genom att språket har inbyggda konstruktioner för att smidigt behandla dem. Listor skrivs med omslutande hakparenteser och varje element åtskiljt av kommatecken. En lista är en ordnad datastruktur, varje element har därför ett index som ni kan använda för att komma åt elementet.

Prova följande:

```
>>> tal = [1, 2, 3, 4, 5]
>>> tal[0]
>>> tal[4]
>>> tal[2]
>>> len(tal)
>>> tal[-1]
>>> tal[1:3]
>>> tal[17]
```

Listor fungerar i stort sett på samma sätt som strängar, ni kan ange både index och slice index för att hämta en del av listan. Ett negativt index innebär att indexeringen sker från slutet av listan istället, vilket även fungerar bra för strängar. Funktionen `len` returnerar längden av en lista eller sträng.

Listor är föränderliga (*mutable*) datatyper. Ni kan förändra en lista genom att ändra värde på olika element, lägga till nya element eller ta bort gamla element. Metoden `append` lägger till ett element i slutet av listan, `insert` infogar ett element på den angivna positionen och skjuter således fram alla efterliggande element en position. Metoden `pop` tar bort ett element ur listan och returnerar elementet, ifall inget index angivets tas sista elementet bort. Slutligen finns `remove` som tar bort den första förekomsten av det element som angivits.

Prova följande:

```
>>> tal[1] = 17
>>> tal.append(63)
>>> tal.insert(0, 100)
>>> tal.pop(1)
>>> tal.remove(3)
>>> help([])
>>> dir([])
```

Tabeller

Ni kan läsa mer om tabeller i kursboken Learning Python, kapitel 8.

Associativa tabeller (*dictionary*, `dict`) är en annan datastruktur som även den har mycket stort inbyggt stöd i Python. Tabeller är föränderliga (*mutable*) likt en lista, men till skillnad från listor är inte en tabell en sekvens av värden och ni kan därför aldrig veta exakt hur Python väljer att lagra er tabell i RAM. Istället för att hämta ut ett element med hjälp av ett index anger ni ett nyckelvärde för det värde ni lägger in i tabellen, sedan kan ni använda er av det för att hämta ut värdet.

Prova följande:

```
>>> a_dict = {'ett': 1, 'två': 2, 'tre': 3, 10: 'tio', 20: 'tjugo'}
>>> a_dict['tre']
>>> a_dict[10]
>>> a_dict['fyra'] = 4
>>> 'tre' in a_dict
>>> 3 in a_dict
>>> help({})
>>> dir({})
```

Kontrollstrukturer

Villkorssatser

Ni kan läsa mer om villkorssatser och sanningstest i kursboken Learning Python, kapitel 12.

En villkorssats evaluerar olika kodstycken beroende på ifall uttrycket är sant eller falskt. Vanligtvis används villkorssatser för att förändra kontrollflödet i ett program eller applikation beroende på yttre parametrar så som indata från användaren.

Skriv följande:

```
>>> def true():
...         print("Evaluerar true()")
...         return True
...
>>> def false():
...         print("Evaluerar false()")
...         return False
...
```

Med hjälp av ovanstående funktioner kommer ni kunna undersöka hur villkorssatser evalueras i Python.

Prova följande:

```
>>> if false() and false():
...       print(”totaluttrycket SANT”)
... else:
...       print(”totaluttrycket FALSKT”)
...
>>> if false() and true():
...       print(”totaluttrycket SANT”)
... else:
...       print(”totaluttrycket FALSKT”)
...
>>> if true() and false():
...       print(”totaluttrycket SANT”)
... else:
...       print(”totaluttrycket FALSKT”)
...
>>> if true() and true():
...       print(”totaluttrycket SANT”)
... else:
...       print(”totaluttrycket FALSKT”)
...
>>> if false() or false():
...       print(”totaluttrycket SANT”)
... else:
...       print(”totaluttrycket FALSKT”)
...
>>> if false() or true():
...       print(”totaluttrycket SANT”)
... else:
...       print(”totaluttrycket FALSKT”)
...
>>> if true() or false():
...       print(”totaluttrycket SANT”)
... else:
...       print(”totaluttrycket FALSKT”)
...
>>> if true() or true():
...       print(”totaluttrycket SANT”)
... else:
...       print(”totaluttrycket FALSKT”)
...
```

Observera att det inte skrivs ut lika många strängar för alla uttryck. Detta beror på att Python är lat i sin evaluering av `and` och `or`. Ifall uttrycket till vänster om `and` är falskt så kommer uttrycket till höger inte att evalueras och hela uttrycket blir således falskt, annars evalueras även uttrycket till höger, och det avgör resultatet av hela uttrycket. Ifall uttrycket till vänster om `or` är sant så kommer uttrycket till höger inte att evalueras och hela uttrycket blir således sant, annars evalueras även uttrycket till höger, och det avgör resultatet av hela uttrycket.

Prova följande:

```
>>> if true() and false() and true():
...       print(”Fall 1 är SANT”)
... elif false() or true() and false():
...       print(”Fall 2 är SANT”)
... elif false() and true() or true():
...       print(”Fall 3 är SANT”)
... elif false() or true() and true():
...       print(”Fall 4 är SANT”)
... else:
...       print(”Fall 5 är SANT”)
...
```

Här kan ni dessutom se en annan konstruktion i Python, `elif`. En `elif`-gren evalueras bara ifall föregående gren i villkorssatsen var falsk.

Upprepningar

Ni kan läsa mer om upprepningar och tekniker där de bör användas i kursboken Learning Python, kapitel 13.

Något ni ofta kommer att använda er av inom programmering är upprepningar, vilket ni kommer göra med loopar. Den allra enklaste loop-konstruktionen i Python är `while`-loopen. `while` är närbesläktad med `if`-satsen, men istället för att köra kodblocket en gång när villkoret är sant så körs kodblocket upprepade gånger så länge villkoret är sant.

Prova följande:

```
>>> counter = 10
>>> while counter > 0:
...       print(”%s är större än noll” %(counter))
...
>>> input = ’’
>>> while input != ’q’:
...       input = raw_input(”Skriv något (’q’ för att avsluta): ”)
...       print(”Du skrev ’%s’.” %(input))
...
```

Iteration

Ni kan läsa mer om iteration i form av for-loopen och tekniker där den bör användas i kursboken Learning Python, kapitel 13.

Eftersom sekvenser (listor, strängar m.m.) är en så vanligt förekommande datatyp finns det givetvis smidiga sätt för att iterera genom och behandla dessa. Att gå igenom en lista med hjälp av en `while`-loop är både osmidigt och en källa till fel, såvida ni inte har en klar förståelse varför ni gör det bör därför `while`-loopen undvikas för att gå igenom sekvenser. `for`-loopen är ett smidigt sätt att iterera genom sekvenser, ni som programmerare slipper tänka på gränser och dylikt, allt sköts via Python.

Prova följande:

```
>>> example = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> index = 0
>>> while index < len(example):
...         element = example[index]
...         print(element)
...         index += 1
...
>>> for element in example:
...         print(element)
...
```

Om ni vill iterera genom en sekvens mellan *n* och *m* finns det dessutom funktioner för detta. Funktionerna `range` och `xrange` genererar en sekvens av siffror beroende på vilka parametrar ni anger.

Prova följande:

```
>>> for number in range(10):
...         print(number)
...
>>> for number in range(5, 11):
...         print(number)
...
>>> help(range)
>>> help(xrange)
```

Inlämningsuppgifter

Uppgift 1a – Summera alla naturliga tal upp till och med 512

Ofta när ni utför beräkningar kommer ni i förväg att kunna räkna ut vilket intervall ni måste gå igenom för att komma fram till svaret. Då är en iteration att föredra eftersom ni med säkerhet kan säga att inga yttre faktorer kommer att påverka resultatet.

Naturliga tal är de heltal som inte är negativa, det vill säga alla tal från 0 och uppåt. Om ni summerar alla naturliga tal upp till och med 128 får ni summan 8256. Er uppgift är att ta reda på vad summan för alla naturliga tal upp till och med 512 är för något.

Uppgift 1b – Produkten av alla positiva heltal upp till och med 512

I denna uppgift ska ni använda er av samma kod som ni skrev i föregående uppgift men nu ska ni göra små ändringar så att ni räknar ut produkten istället för summan.

Produkten för alla positiva heltal upp till och med 12 är 479001600, produkten för alla positiva heltal upp till och med 512 kommer således bli ett fruktansvärt stort tal.

Uppgift 1c – Minsta positiva heltalet som är jämt delbart med siffrorna 1 till och med 13

Ibland är det inte lika lätt att hitta ett intervall att iterera igenom. Istället får ni bestämma er för ett villkor som bara är sant när ni hittat rätt svar och sedan söka igenom ett mycket stort (ibland oändligt) intervall efter rätt svar.

Det minsta positiva heltalet som är jämt delbart med siffrorna 1 till och med 10 är talet 2520. Er uppgift är att ta reda på vilket det minsta talet är som är jämt delbart med alla siffror från 1 till och med 13.

Tips: För att kontrollera ifall ett tal är jämt delbart med ett annat använder ni er av operatorn modulo (`%`). Ifall resultatet av operationen är noll är talen jämt delbara med varandra.

Prova följande:

```
>>> 10 % 5
>>> 10 % 3
```

Uppgift 1d – Summera alla primtal under 1000

Ett primtal är ett naturligt tal som är större än 1 och jämt delbart endast med sig själv och med 1. Att avgöra ifall ett tal är ett primtal eller inte är väldigt kostsamt så ni bör därför tänka efter extra noga när ni formulerar er lösning så att ni inte kontrollerar tal som omöjligt kan vara primtal.

Summan av alla primtal under 100 är 1060. Er uppgift är att ta reda på summan för samtliga primtal under 1000.

Laboration 2 – Funktioner och felsökning

I denna laboration kommer ni arbeta med funktioner och procedurer, något som är en förutsättning för att kunna skriva bra och läsbar kod. Ni kommer att lära er felsöka er egen och andras kod, något som är en förutsättning när ni sedan skriver större program eller applikationer där ni inte alltid kan få en bra överblick över all den kod som finns. Ni kommer även introduceras till enhetstestning, vilket är en metod för att förebygga fel och på så vis minimera den tid ni spenderar med att felsöka er kod.

Övningsuppgifter

Funktioner

Ni kan läsa mer om funktioner och argument i kursboken Learning Python, kapitel 16, 17, 18 och 19.

Nästa logiska steg efter selektion och upprepningar för att strukturera program är funktioner. Ni har redan sett lite av dessa i tidigare laborationer. Funktioner används för att dela in ett större problem i mindre delproblem. Funktioner i Python returnerar alltid ett värde, `None` ifall inget explicit returneras. En funktion kan även acceptera ett antal argument. Argument anges som positionella argument eller som nyckelordsargument.

Prova följande:

```
>>> def my_function(first, second, third, fourth):
...         print(”first = %s, second = %s, third = %s, fourth = %s”
...         %(first, second, third, fourth))
...         return (first + second * fourth) / third
...
>>> my_function(1, 2, 3, 4)
>>> my_function(first = 1, second = 2, third = 3, fourth = 4)
>>> my_function(second = 2, first = 1, fourth = 4, third = 3)
>>> my_function(1, 2, third = 3, fourth = 4)
>>> my_function(1, 2, fourth = 4, third = 3)
```

Positionella argument måste komma i samma ordning som parametrarna i funktionen, ni kan däremot ange nyckelordsargument i en annan ordning men aldrig ange positionella argument efter ett nyckeordsargument. Nyckelordsargument får inte heller upprepa argument som angetts med hjälp av ett positionellt argument tidigare. Slutligen måste alla argument alltid tilldelas ett värde, parametrar som ofta antar samma värde kan däremot ges ett standardvärde. Parametrar med standardvärde måste alltid komma sist i funktionsdefinitionen och inga parametrar utan standardvärde får således föregås av parametrar med standardvärde.

Prova följande:

```
>>> def std_function(first, second, third = 3, fourth = 4):
...         print(”first = %s, second = %s, third = %s, fourth = %s”
...         %(first, second, third, fourth))
...         return (first + second * fourth) / third
...
>>> std_function(1, 2)
>>> std_function(1, 4, fourth = 2)
>>> std_function(5, 0, 5)
```

Felsökning

Den första nivån av felsökning handlar om att granska de felmeddelanden som Python ger i form av undantag. Dessa talar inte bara om vilket fel det var som uppstod, utan även var det uppstod. Ifall flera funktioner anropats på vägen till den position där felet uppstod visas dessutom hela anropskedjan dit i en så kallad traceback, något som underlättar felsökningen betydligt.

Detta är en kort förklaring för några av de vanligaste felen:

- ZeroDivisionError Division med noll, omöjligt då talet blir oändligt stort.
- AttributeError Objektet ni arbetar med har inte den metod ni försöker komma åt.
- ImportError Modulen ni försökt importera går inte att hitta.
- IndexError Ni försöker komma åt ett element i en `list` som inte finns.
- KeyError Ni försöker komma åt ett element i en `dict` som inte finns.
- NameError Variabeln ni använder är inte definierad.
- SyntaxError Ni har stvaat fel.
- TypeError Ni har angett felaktiga argument till en funktion.
- ValueError Ni har angett ett argument med ogiltigt värde till en funktion.

Prova följande:

```
>>> def div(n):
...         return 1.0 / n
...
>>> def fracs(max, min = 0):
...         for num in xrange(min, max):
...                 print(div(num))
...
>>> fracs(10)
```

Förklaring:

- Felet är en division med noll (0).
- Börja med att granska rad 2 i `div`. Här ser ni att anledningen måste vara att `div` anropats med 0 som argument.
- Gå vidare till rad 3 i `fracs`. Här kan ni se att `div` anropas med variabeln `num` som argument. `num` kommer från `xrange(min, max)`. Följer ni traceback ytterligare ett steg bakåt kan ni se att `min` är 0 och `max` är 10, det innebär att `num` kommer vara 0 första iterationen. Problemet är identifierat och en möjlig åtgärd är att byta standardvärdet på `min` från 0 till 1.

Vid mer avancerad felsökning är en debugger det bästa verktyget. Python har en inbyggd debugger som heter `pdb`. `pdb` kan köras antingen från Python-tolken eller som en fristående debugger. För att köra `pdb` från Python-tolken importerar ni modulen `pdb`. När ni använder `pdb` i Python-tolken finns det två stycken funktioner ni kan använda er av; `run` anropas med en kod-sträng som argument och kör sedan kodsträngen i debuggern; `pm` anropas efter att ett fel uppstått för att navigera i anropshistoriken (traceback) och inspektera variabler.

Ni startar `pdb` (som en fristående debugger) med följande kommando från terminalen:
`python -m pdb <filnamn>`

När ni kör `pdb` som en fristående debugger finns det mycket fler kommandon att känna till. Nedan listas de viktigaste, ni kan själva läsa mer i den officiella dokumentationen om `pdb` (http://docs.python.org/library/pdb.html).

Några användbara kommandon:

- Help — Visar hjälp om `pdb` och dess inbyggda kommandon.
- Where — Skriver ut aktuell stacktrace.
- Break — Skapar en stoppunkt.
- Step — Kör nästa rad, gå in i funktionen ifall raden är ett funktionsanrop.
- Next — Kör nästa rad, kör funktionsanropet som en enda rad.
- Return — Kör till slutet av den aktuella funktionen.
- Continue — Kör till nästa stoppunkt, tills ett fel inträffar eller tills koden är slut.

Enhetstestning

När ni arbetar med problemlösning och algoritmer är det viktigt att ni redan från början har en känsla för vilket resultat ni kan förvänta er vid vissa indata. Går det att bryta ner problemet i mindre delproblem är även det ett viktigt steg för att kunna garantera att era delresultat (och således slutresultatet) är korrekta. Ni kommer ofta att förändra era lösningar för att effektivisera dem eller lösa ett specialfall, då är det viktigt att ni kan testa koden så att ni inte av misstag råkar skapa nya problem som annars obemärkt skulle passera.

Ett enhetstest består av ett antal testfall som testar funktionen hos er enhet. Ett testfall i sin tur består av indata till enheten och det förväntade resultatet. Testfallet är inte bara tester av saker som ska fungera utan även saker som inte ska fungera. För att underlätt skrivandet av testfall använder ni ett ramverk, till exempel `PyUnit`. För den här kursen räcker det med något lite enklare och därför finns det ett internt ramverk ni ska använda er av under resten av kursen.

Testmodul

Ni hittar testmodulen i subversion repository för kursen (https://svn-und.ida.liu.se/courses/TDP002/2007-ht1/staff/testdemo/), den innehåller modulen ni använder för att testa er kod samt ett demo som förklarar hur ni ska använda er av modulen.

Körexempel:

```
$ export PYTHONPATH = ”.”
$ python test/demo_test.py
In echo test
In add test
2 tests run (Exceptions: 0, Failures: 0)
```

Demo-modulen `demo.py` ligger i katalogen testdemo. Testmodulen `demo_test.py` ligger i en underkatalog test. För att detta underpaket ska hittas av Python-tolken måste ni först sätta miljövariabeln `PYTHONPATH` till den relativa sökvägen `'.'`. Detta betyder att ni måste stå i testdemo-katalogen när ni kör. I test-katalogen lägger ni all källkod som rör testning. Detta gör att ni får en klar separation mellan den källkod som verkligen behövs för att köra systemet och det som bara används som utvecklingsstöd.

I `test.py` finns ett antal hjälpfunktioner för att göra så kallade `asserts` av olika slag. Genom att använda dessa funktioner får ni kontroll över hur felen uppstår och bra utskrifter var det gått fel (Se `demo_test.py` för enkla exempel hur ni gör detta). Ni bör skriva ett antal tester för varje funktion i modulen som ska testas. Testerna ska prova funktionerna genom att anropa dem med olika möjliga (och omöjliga) argument för att testa om resultatet är det förväntade.

Tillvägagångssätt:

1. För varje modul `<modul>.py` gör ni en test-modul `test/<modul>_test.py`.
2. `<modul>_test.py` importerar både `<modul>.py` och `test.py`.
3. I `<modul>_test.py` gör ni sedan testfunktioner för de olika test ni vill göra. Kalla alla funktioner `test_<namn>`, där `<namn>` är namnet på ert test.
4. Gör en funktion `run_tests` i samma stil som `demo_test` där ni anger de testfunktioner ni vill köra.

Inlämningsuppgifter

Uppgift 2a – Funktioner för ASCII art

Strukturera er lösning på nedanstående problem så att de kan dra nytta av gemensamma lösningar för ett gemensamt delproblem.

Skriv en funktion `frame` som accepterar en textsträng som indata och skriver ut textsträngen inramad med asterisker.

Körexempel:

```
>>> frame(”Välkommen till Python”)
*************************
* Välkommen till Python *
*************************
```

Skriv en funktion `triangle` som accepterar ett heltal som indata och skriver ut en triangel av asterisker med den höjd som heltalet anger.

Körexempel:

```
>>> triangle(3)
  *
 ***
*****
```

Skriv en funktion `flag` som accepterar ett heltal som indata och skriver ut en flagga av asterisker 22 gånger bredare än den storlek som heltalet anger.

Körexempel:

```
>>> flag(1)
**********  **********
**********  **********
**********  **********
**********  **********

**********  **********
**********  **********
**********  **********
**********  **********
```

Uppgift 2b – Inköpslistan

Skriv en uppsättning funktioner ni kan använda från Python-prompten som tillsammans utgör ett program som hanterar en inköpslista.

Körexempel:

```
>>> shopping_list()
1. Kurslitteratur
2. Anteckningsblock
3. Penna
>>> shopping_add()
Vad ska läggas till i listan? Väska
>>> shopping_list()
1. Kurslitteratur
2. Anteckningsblock
3. Penna
4. Väska
>>> shopping_remove()
Vilken sak vill du ta bort ur listan? 2
>>> shopping_list()
1. Kurslitteratur
2. Penna
3. Väska
>>> shopping_edit(2)
Vad ska det stå istället för ”Penna”? Anteckningsblock & Penna
>>> shopping_list()
1. Kurslitteratur
2. Anteckningsblock & Penna
3. Väska
```

Laboration 3 – Algoritmer & kryptering

I denna laboration ska ni arbeta med egna datatyper och kontrollstrukturer på ett sådant sätt att ni kan skriva program som löser riktiga problem. Laborationen är tänkt att öka er förståelse för problemlösning med hjälp av programmering. Även om inte alla algoritmer har ett officiellt namn kan de flesta lösningar på ett problem beskrivas enligt en algoritm.

> *”En algoritm är inom matematiken och datavetenskapen en begränsad uppsättning (mängd) väldefinierade instruktioner för att lösa en uppgift, som från givna utgångstillstånd (starttillstånd) med säkerhet leder till något givet sluttillstånd. Den kan också beskrivas som en systematisk procedur för hur man genom ett begränsat antal steg utför en beräkning eller löser ett problem.” – Wikipedia*

Övningsuppgifter

Abstrakta Datatyper

Hittills har ni bara arbetat med väldigt vanligt förekommande datatyper som strängar och tal men det är inte alltid tal och strängar räcker för att beskriva ett problem på ett bra sätt. *Abstrakta datatyper* (ADT) är ett sätt att beskriva sådana problem som annars inte kan beskrivas med de datatyper som redan finns i språket. Ett spelkort har till exempel två attribut som krävs för att representera kortet (färg och valör). När man hanterar ett sådant spelkort kan man låta en sammansatt datatyp såsom lista eller en tupel representera kortet. Utan funktioner som tydligt förmedlar att man hanterar ett spelkort och inte vilken lista eller tupel som helst kan det dock bli svårt att komma ihåg hur man representerade kort när man står i begrepp att skapa nya och för andra kan det bli svårt att läsa ens program. Därför är det viktigt att skapa en uppsättning *funktioner* som låter en manipulera och visa spelkort.

Vanligen definierar man en abstrakt datatyp genom funktioner för att skapa, modifiera, extrahera data ur och visa datatypen. Finns dessa fyra funktioner är det möjligt att arbeta med er datatyp och utöka den allt eftersom ert program kräver det. Funktioner för att verifiera att data i er datatyp korrekt är även dessa väldigt bra att ha men är inget krav för att ni ska kunna arbeta med er nya datatyp. Som övning inför den här laborationen ska ni skapa två datatyper: spelkort (`card`) och kortlek (`deck`). Nedan anges körexempel på hur datatyperna hanteras. Vi antar att det finns ett kort `three_of_spades` definierat vid det första körexemplets början.

Körexempel:

```
>>> get_value(three_of_spades)

3
>>> get_suit(three_of_spades)

2

>>> display_card(three_of_spades)

”3 of Spades”

>>> card = create_card(13, 2)
>>> display_card(card)
13 of Spades
```

Eftersom ett spelkort är oföränderligt i verkligheten skapar ni inte funktioner för att modifiera det. När ni däremot ska skapa funktioner för att modellera en hel kortlek kommer ni behöva funktioner för att skapa en lek eftersom en kortlek inte alltid behöver innehålla alla spelkort och ordningen på spelkorten kan spela roll beroende på sammanhang. Nedan visas exempel på funktioner för att skapa och extrahera data ur en kortlek. Behövs det fler för att modellera en kortlek? Implementera funktionerna i körexemplet och tänk på huruvida fler funktioner skulle kunna vara användbara när ni implementera algoritmerna i inlämningsuppgifterna.

Körexempel:

```
>>> create_deck()

[’deck’, [(1, 1),(2, 1), ...]]

>>> pick_card(create_deck())

(1, 1)

>>> insert_card(three_of_spades, create_deck())

[’deck’, [(3, 2)(1, 1),(2, 1), ...]]
```

Inlämningsuppgifter

Krypteringsalgoritmen Solitaire (patiens på svenska) utvecklades av Bruce Schneier för tillåta agenter att kommunicera på ett säkert sätt, utan att behöva förlita sig på elektronik eller behöva bära med sig kompromitterande utrustning. Därför behöver ni bara en kortlek för att kryptera eller dekryptera meddelanden, även om det går väldigt mycket enklare med hjälp av ett Python-program!

Uppgift 3a – Generera nycklar inför kryptering

I denna uppgift ska ni implementera algoritmen för `solitaire_keystream` som krävs för att generera den nyckelfras som senare används för att kryptera och dekryptera meddelanden. Notera att ni ska skriva funktioner för att representera en kortlek i Python som en abstrakt datatyp. All hantering av kortleken ska gå genom dessa funktioner. När ni implementerar algoritmen (skriver Pythonkod) för att generera en nyckelfras nedan (`solitaire_keystream`) ska ni tänka att andra personer som endast känner till hur man krypterar med hjälp av en fysisk kortlek ska förstå ert program.

Funktioner för en kortlek (lägg gärna till fler om ni behöver det):

- Skapa en ny och oblandad kortlek (se övningsuppgifterna).
- Blanda och kupera kortleken.
- Flytta ett kort från en position i kortleken till en annan.
- Ta bort ett kort från en viss position ur kortleken.

En kortlek består av 52 spelkort och två jokrar. För enkelhets skull kommer ni bara att använda er av en halv kortlek (två färger) och två jokrar. Varje kort i kortleken tilldelas ett värde. Spelkorten med den första färgen tilldelas värdena 1 till och med 13, spelkorten med den andra färgen tilldelas 14 till och med 26. De två jokrarna tilldelas 27 och 28. Observera att ni ska hantera kortleken som om den bestod av kort, så ni får INTE representera ett kort med enbart ett tal. Däremot bör ni implementera en funktion för att få värdet av ett kort enligt principen ovan.

Algoritm för solitaire_keystream:

Låt den ena jokern heta joker A och den andra joker B.

1. Blanda kortleken inklusive jokrarna.
2. Flytta joker A ner ett steg i kortleken. Ifall jokern redan är längst ner i kortleken flyttas den under det översta kortet.
3. Flytta joker B ner två steg i kortleken. Ifall jokern redan är längst ner i kortleken flyttas den under det näst översta kortet. Ifall jokern är precis ovanför det nedersta kortet i kortleken flyttas den under det översta kortet.
4. Dela kortleken vid de båda jokrarna. Alla kort ovanför den översta jokern flyttas nedanför den understa jokern och alla kort under den understa jokern flyttas ovanför den översta jokern.
5. Flytta lika många kort från övre delen av kortleken som värdet på det understa kortet och sätt in dessa precis ovanför det understa kortet.
6. Det översta kortets värde bestämmer vilket kort i kortleken som ska användas som del i nyckelfrasen. Räkna lika många kort uppifrån som värdet på det översta kortet, titta på kortet direkt efter dessa kort, det är värdet för nästa nyckel i nyckelfrasen, där 1 = A, 2 = B ... 26 = Z. Ifall kortet är en joker blir det ingen nyckel i denna iteration. Notera att detta steg inte förändrar kortleken.
7. Återupprepa från steg 2 tills ni har en nyckelfras med rätt längd.

Körexempel:

```
>>> solitaire_deck = create_deck()
>>> solitaire_keystream(length = 30, deck = solitaire_deck)
MUFKDPSIQMDCCIFLSKENFKSPEFSSUO
```

Uppgift 3b – Kryptera text

I denna uppgift ska ni implementera funktionalitet för att kryptera meddelanden med hjälp av Solitaire.

Algoritm för kryptering:

1. Slopa alla tecken förutom A – Z och konvertera alla gemaner till versaler.
2. Dela upp meddelandet i grupper om fem bokstäver per grupp, använd X för att fylla ut den sista gruppen ifall det skulle behövas. Om vi ska kryptera ordet *Python* blir det PYTHO NXXXX.
3. Använd `solitaire_keystream` för att generera en nyckelfras med samma längd som meddelandet. Anta som exempel att vi får nyckelfrasen ABCDE VWXYZ.
4. Konvertera alla bokstäver i meddelandet (steg 2) till tal, där A = 1, B = 2 ... Z = 26. Exemplet i steg 2 skulle bli 16,25,20,8,15 14,24,24,24,24.
5. Konvertera alla bokstäver i nyckelfrasen (steg 3) till tal, där A = 1, B = 2 ... Z = 26. Exemplet i steg 3 skulle bli 1,2,3,4,5 22,23,24,25,26.
6. Addera talen från meddelandet (steg 4) med talen i nyckelfrasen (steg 5). Subtrahera 26 om summan är 27 eller högre. Exemplen ovan skulle ge 16+1->17, 25+2->1 och så vidare.
7. Konvertera talet (steg 6) till bokstäver.

Körexempel:

```
>>> keystream = solitaire_keystream(30, create_deck())
>>> solitaire_encrypt(”Python”, keystream)
SGUATBCRVV
```

Uppgift 3c – Dekryptera text

I denna uppgift ska ni implementera funktionalitet för att dekryptera meddelanden med hjälp av Solitaire.

Algoritm för dekryptering:

1. Konvertera meddelandet som ska dekrypteras till tal, där A = 1, B = 2 ... Z = 26.
2. Konvertera nyckelfrasen som användes för att kryptera meddelandets tecken till tal, där A = 1, B = 2 ... Z = 26.
3. Subtrahera talen från nyckelfrasen (steg 2) från talen i meddelandet (steg 1). Addera 26 om resultatet är mindre än 1.
4. Konverta siffrorna (steg 3) till bokstäver.

Uppgift 3d – Felsökning av kodfiler

Koden i denna fil (https://svn-und.ida.liu.se/courses/TDP002/2007-ht1/staff/lab2/fel1.py) innehåller tre kodblock som av olika anledningar inte går att exekvera, felsök dem och åtgärda problemet. Se till att lämna kommentarer som markerar var rättningen utförts och vad det var för typ av fel.

Denna fil (https://svn-und.ida.liu.se/courses/TDP002/2007-ht1/staff/lab2/fel2.py) innehåller två testfall där bara det ena fungerar som det ska. Använd er av pdb för att komma fram till varför det andra testfallet inte fungerar som det ska och lägg till kod i programmet på lämplig plats för att åtgärda problemet. Se till att lämna kommentarer där ni gjort ändringar i programmet och beskriv vad det var för typ av fel.

Laboration 4 – Datastrukturer & Interpretatorer

Syftet med denna laboration är att ni ska lära er att skapa modeller av ett scenario för att sedan kunna modifiera och utveckla detta scenario enligt en uppsättning regler som inte får förändras allt eftersom scenariot förändras. Ni kommer dessutom lära er att interpretera och tolka data för att sedan översätta det till ett format som bättre passar ert program.

Inlämningsuppgifter

Sokoban är ett datorspel från Japan där ni kontrollerar en lagerarbetare som ska försöka knuffa alla lådor till sina rätta platser i ett varuhus. Reglerna är enkla och det finns inte särskilt många olika typer av objekt att hålla reda på. Lagerarbetaren kan röra sig upp, ner, till vänster och till höger. Han kan knuffa lådor som är framför honom så länge som det inte är en vägg eller en annan låda bakom den låda han knuffar. Spelet slutar först då alla lådor är på rätt plats.

Uppgift 4a – Datastrukturer och funktioner för att representera spelplanen

När det gäller att representera en samling objekt eller ett scenario finns det två olika tillvägagångssätt. Det första sättet är att se det hela som ett scenario, i detta fall skulle det betyda att spelplanen är det viktiga och alla händelser i programmet förändrar en detalj på spelplanen. Det andra sättet är att se varje enskilt objekt på spelplanen som en fristående del och det är först när ni använder er av alla delarna tillsammans som en spelplan uppstår. Hur ni väljer att arbeta är givetvis upp till er och det finns givetvis fler varianter där ni tar lite från båda metoderna.

I denna uppgift ska ni skriva funktioner för att bygga upp och förändra en spelplan. Det ska finnas möjlighet att definiera utseendet på spelplanen. Det ska gå att placera ut lådor och lagerplatser på spelplanen. Det ska gå att förflytta en lagerarbetare och knuffa lådor. Utöver detta ska ni dessutom skriva en funktion `sokoban_display(...)` som ritar ut spelplanen på ett snyggt sätt.

Följande objekt ska kunna finnas på spelplanen:

- @ Lagerarbetaren
- O Lådorna
- # Väggar
- <space> Golvyta
- . Lagringsplats
- * Låda som står på en lagringsplats
- + Lagerarbetare som står på en lagringsplats

Körexempel:

```
>>> soko_walls.append(create_wall(x = 1, y = 0)
>>> soko_walls.append(create_wall(x = 0, y = 1)
>>> soko_walls.append(create_wall(x = 2, y = 1)
>>> soko_walls.append(create_wall(x = 1, y = 2)
>>> soko_player = move_player(create_player(), 1, 1)
>>> soko_walls
[['wall', 1, 0], ['wall', 0, 1], ['wall', 2, 1], ['wall', 1, 2]]
>>> soboban_display(wall = soko_walls, player = soko_player)
 #
#@#
 #
```

Uppgift 4b – Ladda spelplanen från fil

Sokoban är ett populärt spel och det finns många officiella nivåer. Några av dessa nivåer hittar ni i körexemplet nedan. Er uppgift är att skriva funktioner för att ladda in en spelplan från fil så att ni kan använda den tillsammans med de funktioner ni tidigare skrivit för att representera Sokoban.

Ni kan hitta de nivåer som är beskriva nedan på kurskontot för kursen (http://www.ida.liu.se/~TDP002/resources/lab4/). Där finns dels några nivåer som har filnamn som slutar på `.sokoban` men även en samling nivåer i filen `sokoban_levels.txt`. Ur filen med flera banor kan ni kopiera banor till enskilda filer, alternativt kan ni göra egna.

Körexempel:

```
>>> first_level = sokoban_load('first_level.sokoban')
>>> sokoban_display(first_level)
    #####
    #   #
    #o  #
  ###  o##
  #  o o #
### # ## #   ######
#   # ## #####  ..#
# o  o          ..#
##### ### #@##  ..#
    #     #########
    #######
>>> second_level = sokoban_load('second_level.sokoban')
>>> sokoban_display(second_level)
############
#..  #     ###
#..  # o  o  #
#..  #o####  #
#..    @ ##  #
#..  # #  o ##
###### ##o o #
  # o  o o o #
```

```
#     #      #
############
```

Uppgift 4c – Kollisioner

Innan ni kan sätta ihop de olika delarna till ett fullt fungerande spel måste ni se till att lagerarbetaren kan förflytta sig i rummet men inte kunna gå igenom väggar eller lådor. Beroende på er tidigare implementation av spelplanen finns det lite olika möjligheter till kontroller men det är väldigt viktigt att ni tänker igenom alla möjligheter så att det fungerar bra i alla olika riktningar.

Er uppgift är att skriva en funktion `player_can_move(...)` och en funktion `crate_can_move(...)` som returnerar sant ifall det är möjligt att flytta en spelare eller en låda till en ny position och falskt ifall det skulle innebära en kollision.

Uppgift 4d – Sokoban

Nu är det dags att sätta ihop de funktioner ni skrivit till ett spel! Ni ska implementera ett användargränssnitt så att en spelare kan navigera lagerarbetaren genom spelplanen och knuffa lådor till sina rätta platser. När spelaren har löst nivån ska spelet avslutas. De olika nivåerna ska laddas från en katalog `'levels'` och programmet ska söka igenom katalogen efter nya nivåer vid varje körning.

Körexempel:

```
$ python sokoban.py
Welcome to Sokoban, please choose a level:
1. first_level
2. second_level
3. my_level
Choose: 3
#####
#.O@#
#####
Make your move (l)eft, (r)ight, (u)p, (d)own: l
#####
#*@ #
#####
Congratulations, You completed level 'my_level'!
```

Laboration 5 – Funktioner av högre ordningen

Syftet med denna laboration är att ni ska lära er abstrahera problem till en sådan nivå att ni kan återanvända funktioner och kodstycken till snarlika uppgifter. På så vis kommer ni spendera mindre tid till att kopiera och klistra in och desto mer tid till att faktiskt programmera, det kommer även resultera i en mindre kodbas vilket alltid är bra ifall ni måste underhålla den kod ni skrivit.

Övningsuppgifter

List Comprehension

Ni kan läsa mer om List Comprehension och generatorer i kursboken Learning Python, kapitel 14 & 20.

List comprehension definierar en lista med hjälp av operationer på en genererad lista. Syntaxen för list comprehension påminner om hur listor definieras genom uppräkning, men är mer generell.

Prova följande:

```
>>> [number for number in range(10)]
>>> sum([number for number in range(513)])
>>> ['*' * (star * 2 + 1) for star in range(4)]
```

Ni kan även använda er av villkorssatser inne i en list comprehension för att på så vis filtrera ut de värden som är relevanta för ert resultat.

Prova följande:

```
>>> def is_prime(n):
...         for m in xrange(2, n):
...                 if n % m == 0: return False
...         return True
...
>>> sum([number for number in range(1000) if is_prime(number)])
```

Listan baseras på uttrycket innan `for`, variabeln direkt efter `for` antar successivt alla värden i den sekvens (`range`) ni anger. Uttrycket till vänster om `for` kan vara vilket uttryck som helst, även om det bör returnera ett värde. Vill ni bara ta med delar av den sekvens som angivits baserat på ett villkor kan ni placera en `if`-sats i slutet av list comprehension.

Filter

Funktionen `filter` filtrerar en lista med värden beroende på resultatet av en funktion. Villkorssatsen i en list comprehension är i stort sett en implementation av funktionen `filter`, även om syntaxen är något annorlunda.

Prova följande:

```
>>> def is_prime(n):
...         for m in xrange(2, n):
...                 if n % m == 0: return False
...         return True
...
>>> sum(filter(is_prime, range(1000)))
```

`filter` kan även ses som en sökning, funktionen i det första argumentet är sökvillkoret, listan i det andra argumentet är databasen ni vill söka i och resultatet är alla element i databasen som matchade sökvillkoret.

Map

Ett ofta förekommande inslag i programmering är att utföra samma operation på varje element i en lista. Detta brukar kallas för map inom funktionell programmering. Funktionen map finns i Python och tar (minst) två argument. Resultatet är en lista med returvärden från funktionen då funktionen anropas en gång för varje element i listan.

Prova följande:

```
>>> def is_prime(n):
...         for m in xrange(2, n):
...                 if n % m == 0: return False
...         return True
>>> map(is_prime, range(10))
```

Lambda

Ni kan läsa mer om lambda och varför ni bör använda det i kursboken Learning Python, kapitel 19.

`lambda`-funktioner är små, simpla och namnlösa funktioner som definieras medans programmet eller applikationen körs. Kroppen i en `lambda`-funktion kan bara innehålla ett uttryck (en kodrad).

Prova följande:

```
>>> sqrt = lambda x: x ** 0.5
>>> sqrt(2)
>>> pow = lambda x, y = 2: x ** y
>>> pow(2)
>>> pow(2, 10)
```

Inlämningsuppgifter

Uppgift 5a – Tillbaka till Laboration 1

Det är dags att reflektera över vad ni har lärt er under kursens gång och se ifall det finns utrymme att förbättra era tidigare lösningar. I laboration 1 skrev ni två olika funktioner för att få summan av ett intervall och en annan funktion för att få produkten av ett intervall. Ni skrev även en funktion för att hitta det minsta talet som är delbart med samtliga tal från 1 till och med 13.

Er uppgift är att skriva om uppgift 1a och 1b så att ni kan lösa båda problemen med hjälp av en funktion och ett lambda uttryck. Ni ska dessutom skriva om uppgift 1c så att ni inte behöver belasta processorn med en lång iteration. Med hjälp av primtalsfaktorisering och list comprehensions går det snabbt och enkelt att lösa denna uppgift även ifall intervallet hade varit större.

Uppgift 5b – Musiksamlingen

En gång för länge sedan i en värld långt långt borta gick det inte att streama musik från internet. Er uppgift är att skriva en liten databas för att hantera en musiksamling och jämföra den med andra samlingar.

Funktioner för att hantera musiksamlingen:

- `subsume(dict1, dict2)` returnerar `True` om och endast om `dict1` innehåller `dict2` som en delstruktur, annars `False`.
- `restrict_with(dict_list, restrict_list)` tar bort alla element i `dict_list` som även finns i `restrict_list`.
- `unify(dict1, dict2)` returnerar unionen av de båda tabellerna ifall de har kompatibla nycklar och det går att slå ihop `dict1` med `dict2`. I annat fall returneras `None`.
- `subsume_list(dict_list, subs_list)` returnerar `True` ifall varje element i `dict_list` är en delstruktur av något element i `subs_list`.

Körexempel:

```
# {'a': 1} är en delmängd av {'a': 1, 'b': 2}
>>>  subsume({'a': 1, 'b': 2}, {'a': 1})
True
# {'d': 5} är inte en delmängd av {'c': 3, 'd': 4}
>>> subsume({'c': 3, 'd': 4}, {'d': 5})
False
# {'g': 7} är inte en delmängd av {'e': 6, 'f': 7}
>>> subsume({'e': 6, 'f': 7}, {'g': 7})
False

# {'b': 2} tas bort ur dict_list då den specifieras i restrict_list
>>> dlist = [{'a': 1}, {'b': 2}]; rlist = [{'b': 2}]
>>> restrict_with(dlist, rlist)
>>> dlist
[{'a': 1}]
```

```
>>> unify ({'a': 1, 'b': 2}, {'c': 3, 'd': 4})
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
>>> unify ({'a': 1, 'b': 2}, {'b': 2})
{'a': 1, 'b': 2}
>>> unify ({'a': 1, 'b': 2}, {'b': 3})
None

>>> subsume_list ([{'a': 1}], [{'a': 1, 'b': 2}, {'c': 3}])
True
>>> subsume_list ([{'a': 1}, {'b': 2}], [{'a': 1, 'b': 2, 'c': 3}]
True
>>> subsume_list ([{'a': 1}, {'b': 2}], [{'a': 1, 'c': 3}])
False
```

Uppgift 5c – Företagsdatabasen

En databas över personer som arbetar på ett företag definieras som en lista med tabeller. Skriv en funktion som söker i databasen. Första argumentet till funktionen är databasen som ska genomsökas, andra argumentet anger vilket fält matchningen ska ske emot och tredje argumentet anger vilket värde det fältet ska ha. Returvärdet ska vara en lista med de poster i databasen som matchar sökningen.

Körexempel:

```
>>> db = [
          {'name': 'Jakob', 'position': 'assistant'},
          {'name': 'Åke', 'position': 'assistant'},
          {'name': 'Ola', 'position': 'examiner'},
          {'name': 'Henrik', 'position': 'assistant'}
          ]
>>> dbsearch(db, 'position', 'examiner')
[{'position': 'examiner', 'name': 'Ola'}]
```

Uppgift 5d – Needles and Haystacks

Skriv en funktion som tar reda på om en viss lista innehåller ett visst element.

Körexempel:

```
>>> haystack = 'Can you find the needle in this haystack?'.split()
>>> contains('find', haystack)
True
>>> contains('needle', haystack)
True
>>> contains('haystack', haystack)
False
```

Uppgift 5e – Funktioner som indata

Skriv en listgenererande funktion, `generate_list`. Funktionen ska ta en funktion som första argument och ett heltal som andra argument. Funktionen som angetts som första argument ska därefter anropas så många gånger som heltalet anger, med ett ökande siffervärde som enda parameter varje gång. Resultatet av funktionen ska läggas i en lista som slutligen returneras från `generate_list`.

Körexempel:

```
>>> def mirror(x): return x
>>> generate_list(mirror, 4)
[0, 1, 2, 3]
>>> def stars(n): return '*' * n
>>> generate_list(stars, 5)
['', '*', '**', '***', '****']
```

Uppgift 5f – Kommandoskal

Den här uppgiften går ut på att konstruera ett simpelt kommando-system där användaren presenteras med en kommando-prompt och kan skriva in kommandon som sedan tolkas via ett callback system. Systemet ska vara ganska likt en terminal, fast begränsat. Notera att även detta är en interpretator likt uppgift 4b och även om uppgifterna till en början förefaller helt olika varandra kommer ni snart märka att skillnaderna är väldigt små.

Följande kommandon ska finnas tillgängliga:

- pwd — Skriva ut nuvarande arbetskatalog.
- cd — Byt nuvarande arbetskatalog.
- ls — Lista innehållet i nuvarande arbetskatalog.
- cat — Skriv ut innehållet i en fil.

Körexempel:

```
command> ls
core.py
interpretor.py
command> cd ..
command> pwd
/home/pyuser/code/python/
command> ls
command_system/
README
web-project/
command> cat README
These are my Python projects.

command_system – A simple interpreter based on a callback system.
web-project – A site where I can display my projects.
command> cd web-project
command> ls
index.py
command>
```

Uppgift 5g – Partial

Skriv en funktion partial som tar en annan funktion och ett värde som indata. Partial ska returnera en ny funktion som gör samma sak som den angivna funktionen men där första argumentet till den angivna funktionen har fått det värde som angavs som andra argument.

Körexempel:

```
>>> def add(n, m): return n + m
...
>>> add_five = partial(add, 5)
>>> add_five(3)
8
>>> add_five(16)
21
```

Uppgift 5h – Compose

Skriv en funktion compose som tar två funktioner som indata och returnerar en funktion som innebär att utdata från den andra funktionen blir indata till den första.

Körexempel:

```
>>> x = compose(f, g)
>>> x(5) == f(g(5))
True
>>> def multiply_five(n):
...         return n * 5
...
>>> def add_ten(x):
...         return x + 10
...
>>> composition = compose(multiply_five, add_ten)
>>> composition(3)
65
>>> another_composition = compose(add_ten, multiply_five)
>>> another_composition(3)
25
```

Uppgift 5i – Filter the mapped result

Skriv en funktion `make_filter_map` som tar en `filter`-funktion och en `map`-funktion som argument. Funktionen ska returnera en funktion som tar en lista som argument och applicerar `map`-funktionen på varje element i listan som `filter`-funktionen är sann för. `make_filter_map` ska använda funktionerna `partial` och `compose` från tidigare uppgifter för att sätta ihop funktionerna `map` och `filter` med indata-funktionerna till det önskade resultatet.

Körexempel:

```
>>> process = make_filter_map(lambda x: x % 2 == 1, lambda x: x * x)
>>> process(range(10))
[1, 9, 25, 49, 81]
```

Laboration 6 – Sökning, sortering och riktiga program

Syftet med denna laboration är att ni ska få en bättre förståelse för vilket arbete som krävs av processorn och hur ni kan minimera den tid era program arbetar med att gå igenom listor eller söka efter data. Slutligen ska ni skriva några riktiga program som ni faktiskt kan ha användning för senare i er utbildning.

Inlämningsuppgifter

Uppgift 6a – Linjärsökning

Linjärsökning är den simplaste formen av sökning ni kan tänka er. Det går helt enkelt ut på att ni börjar söka från början av listan och slutar när ni hittar elementet ni letar efter eller kommer till slutet av listan. Linjärsökning har fördelen att den även fungerar för osorterad data.

Skriv en funktion `linear_search` som söker igenom en lista (`haystack`) efter ett specifikt värde (`needle`). Funktionen ska dessutom ha möjlighet att ta ett tredje argument med en funktion för att specificera hur jämförelsen ska gå till.

Körexempel:

```
>>> linear_search(imdb, 10, lambda e: e['score'])
{'title': 'Raise your voice', 'actress': 'Hilary Duff', 'score': 10}
```

Uppgift 6b – Binärsökning

Binärsökning fungerar om mängden ni söker i är sorterad och på så sätt att ni hela tiden jämför det eftersökta elementet med elementet på mittenpositionen i sökmängden, därefter begränsar ni sökmängden beroende på utfallet av jämförelsen.

Skriv en funktion `binary_search` som söker igenom en lista (`haystack`) efter ett specifikt värde (`needle`). Funktionen ska dessutom ha möjlighet att ta ett tredje argument med en funktion för att välja ut det som ska jämföras.

Uppgift 6c – Insertion sort

Ofta är det bra att kunna sortera data så att det går snabbare att söka igenom den. Skriv en funktion `insertion_sort` som tar en lista med element och en funktion för att specificera det man ska sortera med avseende på. Information om hur den fungerar hittar ni på http://en.wikipedia.org/wiki/Insertion_sort.

Körexempel:

```
>>> db = [
          ('j', 'g'), ('a', 'u'), ('k', 'l'), ('o', 'i'),
          ('b', 's'), ('@', '.'), ('p', 's'), ('o', 'e')
          ]
>>> insertion_sort(db, lambda e: e[0])
>>> db
[('@', '.'), ('a', 'u'), ('b', 's'), ('j', 'g'), ('k', 'l'), ('o',
'e'), ('o', 'i'), ('p', 's')]
```

Uppgift 6d – Quicksort

Skriv en funktion `quicksort` som tar en lista med element och en funktion för att specificera det man ska sortera med avseende på. Information om hur den fungerar hittar ni på http://en.wikipedia.org/wiki/Quicksort.

Körexempel:

```
>>> db = [
          ('j', 'g'), ('a', 'u'), ('k', 'l'), ('o', 'i'),
          ('b', 's'), ('@', '.'), ('p', 's'), ('o', 'e')
          ]
>>> quicksort(db, lambda e: e[0])
>>> db
[('@', '.'), ('a', 'u'), ('b', 's'), ('j', 'g'), ('k', 'l'), ('o',
'e'), ('o', 'i'), ('p', 's')]
```

Uppgift 6e – Copyright

I den här uppgiften ska ni lägga till copyright-information i källkodsfiler. Ni ska dessutom enkelt kunna byta ut copyright-informationen till en annan ifall ni vill publicera koden under olika licensvilkor.

Skriv ett program som kan användas på alla era källkodsfiler för att infoga copyright-information i filen. Informationen ska infogas mellan markörerna `BEGIN COPYRIGHT` och `END COPYRIGHT`. Ifall dessa markörer inte finns i filen ska filen inte förändras. Ifall markörerna förekommer på flera ställen i filen ska copyright-informationen infogas på varje plats i filen. Ifall det redan finns information mellan markörerna i filen ska den informationen ersättas. Markörerna ska vara kvar i den resulterande filen så att ni senare kan använda samma program för att byta till annan information.

Programmet ska acceptera två kommandoradsparametrar. Den ena ska vara en fil som innehåller copyright-informationen och den andra ska vara den fil eller katalog där informationen ska infogas. Ifall det är en katalog ska samtliga filer i katalogen påverkas. Programmet ska även acceptera en flagga för att bara en viss filändelse ska behandlas och en flagga för att ändra filändelsen på resultatfilen.

Uppgift 6f – Säkerhetskopiering och backup

I den här uppgiften ska ni skriva ett enkelt system för säkerhetskopiering. Systemet ska ha olika rutiner för olika typer av filer. Därför ska ni använda er av en konfigurationsfil där varje sektion motsvarar en filtyp. Ni ska kunna ange vilken filändelse filerna ni vill säkerhetskopiera har, vilken källkatalog som ska säkerhetskopieras samt vart säkerhetskopian ska lagras.

Skriv ett program som tar en filtyp som parameter. Programmet ska sedan läsa in vad filtypen innebär för filer och säkerhetskopiera dessa.

För att själva säkerhetskopieringen inte ska jobba i onödan ska ni använda ett program som heter `rsync`. `rsync` är inte en Python-modul utan ett program som bara kopierar skillnaden mellan källfilen och målfilen (ifall målfilen redan finns).