
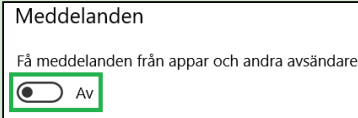


Slutavstämning Programmering 1 - Del 1

28 maj 2020

<table><tr><td>Tidpunkter</td><td>Tid för Del 1: 180 minuter</td></tr><tr><td>9:00</td><td>Starttid Del 1</td></tr><tr><td>11:45</td><td>Rekommenderad senast tid att lämna in på Classroom för att undvika teknikstrul.</td></tr><tr><td>12:00</td><td>Sluttid för första delen, inlämning Classroom.</td></tr><tr><td>12:05</td><td>Inlämning stängs. Kod som lämnas in efter denna tid accepteras ej. OBSERVERA: Kod från Del 1 som lämnas in efter 12:05 bedöms ej.</td></tr><tr><td>13:15</td><td>Samling i Meet för Del 2.</td></tr></table>	Tidpunkter	Tid för Del 1: 180 minuter	9:00	Starttid Del 1	11:45	Rekommenderad senast tid att lämna in på Classroom för att undvika teknikstrul.	12:00	Sluttid för första delen, inlämning Classroom.	12:05	Inlämning stängs. Kod som lämnas in efter denna tid accepteras ej. OBSERVERA: Kod från Del 1 som lämnas in efter 12:05 bedöms ej.	13:15	Samling i Meet för Del 2.	<p>Praktisk information</p> <p>Under hela provet skall du vara ansluten till google-meet med din telefon.</p> <p>Telefonen ska vara uppställd bakom dig på ett sådant sätt att provvakten ser både dig och din skärm tydligt.</p> <p>Din dator ska vara bortkopplad från Internet och notifikationer avstängda.</p> <p>Är din dator ansluten med nätverkskabel räcker det inte att sätta datorn i flygplansläge, den är då fortfarande ansluten till Internet.</p> <p>För att stänga av notifikationer: Inställningar ⇒ System ⇒ Meddelanden och åtgärder ⇒</p> <p>Hörlurar eller öronsnäckor är inte tillåtet under provet.</p>  
Tidpunkter	Tid för Del 1: 180 minuter												
9:00	Starttid Del 1												
11:45	Rekommenderad senast tid att lämna in på Classroom för att undvika teknikstrul.												
12:00	Sluttid för första delen, inlämning Classroom.												
12:05	Inlämning stängs. Kod som lämnas in efter denna tid accepteras ej. OBSERVERA: Kod från Del 1 som lämnas in efter 12:05 bedöms ej.												
13:15	Samling i Meet för Del 2.												
<p>När du känner dig redo att lämna in Del 1, gör så här:</p> <ol style="list-style-type: none">(1) Spara alla filer.(2) Zippa mappen där alla filerna finns. Se till så att alla filer är med.(3) Döp zip-filen till: namn_efternamn_avstämning7_del1.zip(4) Sätt igång internet(5) Öppna webbläsaren, går direkt in på google classroom!(6) Skicka in zip-filen på uppgiften "Slutavstämning del 1".(7) Markera uppgiften som inlämnad ← OBS: Glöm inte detta!	<p>Lite blandat:</p> <p>Längst ner i detta dokument finner du Ruby Cheat Sheet. Har du en utskriven kopia får du använda både den digitala (i detta dokumentet) och den utskrivna om du vill.</p> <p>På vissa uppgifter finns det "syntax-tips" för att ge tips. Har du en lösning som är lika lätt eller lättare utan att använda syntax-tipsen så är det helt okej använda din lösning.</p> <p>När vi skriver att du inte behöver validera input kan du förutsätta alla argument och inputdata är av "rätt sort". När vi skriver att du ska validera input så ska ske på samtliga ställen där det är rimligt. Du får använda dig av if-satser, undantagshantering eller en kombination av dessa. För att visa A-nivå i validering på provet som helhet behöver du vid någon uppgift visa färdighet i undantagshantering.</p> <p>Sista uppgiften i Del 1 är en utvärdering som du kan skriva om du har tid kvar. Om du inte hinner skriva den finns det även chans att skriva en utvärdering i Del 2 av provet.</p>												
<p>VIKTIGAST AV ALLT</p> <p>Även om du inte klarar alla delar av en uppgift finns det möjligheter att visa på alla nivåer, beroende vilka moment du klarar av. Gör så gott du kan, och tänk på att lämna in ALL KOD, även om du inte känner att den blev helt färdig. Tro på dig själv och din förmåga!</p>													

Bedömningsmatris för avstämningen (Del 1 och Del 2 som helhet)

	C	A
Verktyg och syntax	Du hanterar med viss säkerhet programmatiska verktyg och syntax.	Du hanterar med säkerhet programmatiska verktyg och syntax.
Felsökning	Du hanterar felsökning av din kod. Till exempel genom att beskriva det i utvärderingen.	I uppgiften som handlar om felsökning visar du på strukturerad felsökning av kod.
Tydlighet och dokumentation	Du skriver tydlig och lättläst kod samt gör noggrann kommentering av din kod.	Du skriver tydlig och lättläst kod samt visar färdighet i att dokumentera kod på ett strukturerat sätt, enligt den struktur vi gått igenom i kursen.
Interaktion med användaren samt validering	Du skriver kod som med viss säkerhet är anpassad för att interagera med den avsedda användaren. Det kan till exempel handla om validering av data samt tydlig interaktion med användaren.	Du skriver kod som med säkerhet är anpassad för att interagera med den avsedda användaren. Det kan till exempel handla om validering av data samt tydlig interaktion med användaren. Du visar färdighet i undantagshantering.
Den färdiga koden	Du skriver kod som löser programmatiska problem med tillfredsställande resultat.	Du skriver kod som löser avancerade programmatiska problem med gott resultat.
Utvärdering	Du utvärderar med nyanserade omdömen hur arbetet har gått.	Du utvärderar med nyanserade omdömen hur arbetet har gått. Du ger även förslag på förbättringar.
Datavetenskapliga begrepp	I din kommunikation kring uppgiften använder du med viss säkerhet datavetenskapliga begrepp.	I din kommunikation kring uppgiften använder du med säkerhet datavetenskapliga begrepp.

1) addition

- Skriv en funktion i Ruby med namnet **addition**. Skriv funktionen i filen **1_addition.rb**, som finns i samma mapp som detta dokument.
- Funktionens **argument** ska vara en **sträng** bestående av två positiva heltal med ett additionstecken mellan.
- Funktionen ska **returnera** summan av heltalen som en **integer**.
- Dokumentera din kod, se gröna rutan längst ner för vad som bedöms i uppgiften.
- **Du behöver inte validera input** i denna uppgift. Du kan alltså förvänta dig att argumentet till funktionen är av rätt sort.

Exempel på funktionsanrop och returnerad integer:

```
addition("5+3")           #=> 8
addition("237+100")        #=> 337
```

Tips: I denna uppgift kan metoden **.split(tecken)** användas för att dela upp en sträng utifrån ett **tecken**. Se exempel:

Exempel 1:

```
"3+20".split("+")         #=> ["3", "20"]
```

Exempel 2:

```
string_with_numbers_and_addtion = "77+1234"
array_with_numers_as_string_elements = string_with_numbers_and_addtion.split("+")    #=> ["77", "1234"]
```

Vad ska jag fokusera på i denna uppgift?

I denna uppgift ligger fokus på **dokumentation av din kod** vad gäller A-nivå. Dokumentera koden på ett strukturerat sätt, på det sätt vi har gått igenom i kursen. Du behöver inte kommentera löpande i koden. Tänk på att använda tydliga variabelnamn.

Själva kodningen ligger på en ungefärlig C-nivå. Men där ser vi även en helhet i samband med de andra uppgifterna.

2) division

- Skriv en funktion i Ruby med namnet **division**. Skriv funktionen i filen **2_division.rb**, som finns i samma mapp som detta dokument.
- Funktionens **argument** ska vara en **sträng** bestående av två positiva integers med ett divisionstecken mellan.
- Funktionen ska **returnera** en **array** med två **integers** som element. Den första integern ska stå för hur många hela gånger divisionen går ut och den andra integern ska stå för vilken som blir resten i divisionen. Se exempel:

Exempel på funktionsanrop och returnerad array:

```
division("9/4")           #=> [2, 1]
division("27/5")          #=> [5, 2]
division("16/4")          #=> [4, 0]
```

- Validera i koden om användaren anropar funktionen med något annat än en sträng. Se gröna rutan längst ner för vad som bedöms i uppgiften.
- **Du behöver inte dokumentera eller kommentera** denna uppgift. Tänk dock på att ha tydliga variabelnamn.

Tips: I denna uppgift är modulo-operatorn % användbar, som returnerar rest vid division. Som i förra uppgiften går det även bra att använda .split.

```
16 % 4 #=> 0
```

```
17 % 4 #=> 1
```

Testa i IRB om du är osäker på hur % fungerar.

Vad ska jag fokusera på i denna uppgift?

I denna uppgift kan du visa **validering av input**, vad gäller om användaren anropar funktionen med något annat än en sträng. Tänk på tydliga felmeddelanden. För att visa A-nivå i validering på provet **som helhet** behöver du vid **någon uppgift** visa färdighet i undantagshantering. Detta kan du göra i denna uppgift, men även i andra uppgifter.

Själva kodningen ligger på en ungefärlig C-nivå. Men där ser vi även en helhet i samband med de andra uppgifterna.

3) calculator

Skriv ett program i Ruby enligt nedanstående instruktion. Skriv programmet i filen **3_calculator.rb** som finns i samma mapp som detta dokument.

- Programmet ska med hjälp av en **input-loop** ge användaren möjlighet att skriva in en **sträng** bestående av två positiva heltal med antingen ett additionstecken eller ett divisionstecken emellan.
- Programmet ska sedan **skriva ut på skärmen** svaret på operationen. Om det är en division ska svaret vara formulerat som en array, på samma sätt som i uppgift två.
- Användaren ska sedan kunna skriva in en ny input och få en ny utskrift på skärmen. Användaren ska även kunna stänga av programmet med lämpligt input.
- **Du behöver inte validera input i denna uppgift.** Du kan med andra ord anta att användaren ger den sorts input som är tänkt.
- **Du behöver inte dokumentera eller kommentera din kod.** Tänk dock på att ha tydliga variabelnamn.

Vad ska jag fokusera på i denna uppgift?

I denna uppgift ligger fokus på tydlig **interaktion med användaren** vad gäller A-nivå.
Ditt mål är alltså att skriva ett program där det är **tydligt** för användaren hur man använder programmet.
Samt att utskrifterna på skärmen är tydliga för användaren att tolka.

Själva kodningen ligger på en ungefärlig C-nivå. Men där ser vi även en helhet i samband med de andra uppgifterna.

Se nästa sida för tips till uppgift 3!

Tips till uppgift 3

Tips #1:

Rekommendation är att i denna uppgift använda dig av funktionerna i uppgift 1 och uppgift 2.

För att läsa in kod från en annan ruby-fil kan du använda kommandot **require_relative**, se exempel:

```
require_relative "min_kod.rb"    #Läser in innehåll från filen med relativ sökväg "min_kod.rb"
```

Tips #2:

Om du inte har fått funktionerna i uppgift 1 och uppgift 2 att fungera, kan du använda dig av följande *låtsasfunktioner* av addition och division. Det som bedöms i uppgift 3 är kodningen av inputloopen samt tydlig interaktion med användaren, vilket du fortfarande kan visa i detta fall.

```
def fake_addition(string_with_two_numbers_and_addition)
  return 1
end
```

```
def fake_division(string_with_two_numbers_and_division)
  return [1,1]
end
```

Tips #3:

Du får använda kommandot **.include?(tecken)** i denna uppgift. Exempel på hur kommandot fungerar:

```
favoritmat = "varmkorv"
favoritmat.include?("m")    #=> true
favoritmat.include?("p")    #=> false
```

4) Strukturerad felsökning

Se filen `4_add_zeros_to_array.rb` som finns i samma mapp som detta dokument. Filen innehåller kod som är början på en funktion i Ruby. Funktionen ska ta som argument en *array* samt en *integer*. Funktionen ska lägga till samma antal nollor som integern i arrayen. Sedan ska arrayen returneras. Koden i filen är ofullständig och innehåller fel. Din uppgift är skriva färdigt funktionen **samt göra en strukturerad felsökning steg för steg**.

Instruktioner

- Med utskrifter på lämpliga ställen i koden ska du **steg för steg** identifiera saker som fungerar eller inte fungerar.
- Mellan varje steg gör du en kopia av koden och kommenterar bort den tidigare kopian. Viktigt är att **alla versioner i din testning finns med** i dokumentet så att man kan följa testningens steg. Gör på samma sätt som vi gått igenom under repetitionen till detta prov.
- Redovisa med kommentarer vad i din funktion du testat, samt vilket resultat testningen gav.

Exempel:

```
# Förra testet visade att... Nu testat jag följande: ...  
# Kod kod kod kod kod kod kod kod kod kod kod kod kod  
# Kod kod kod kod kod kod kod kod kod kod kod kod kod  
# Kod kod kod kod kod kod kod kod kod kod kod kod kod
```

- Visa flera steg i felsökningen, även om du ser hela lösningen från början. **Ha med minst tre steg** i din felsökning.

Tips: Gör så här för att på ett enkelt sätt kommentera bort (samt ta tillbaka från kommenterat läge) kod i Visual Studio Code:

1) Markera koden du vill kommentera bort. 2) Klicka på menyn Edit. 3) Välj "Toggle line comment".

Vad ska jag fokusera på i denna uppgift vad gäller bedömning?

I denna uppgift ligger fokus på **strukturerad felsökning** vad gäller A-nivå.

Ursprungskod till uppgift 4

(samma som från början finns i filen `4_add_zeros_to_array.rb`, om du skulle råka radera den)

```
def add_zeros_to_array(input_array, number_of_zeros)

  i = 0

  while i >= input_array.length

    i + 1

    input_array

  end

  return

end
```


5) OM DU HAR TID ÖVER: Utvärdering av uppgift 3

- Denna uppgift behöver du inte göra om du har lite tid kvar. Det finns även en chans att skriva en utvärdering i nästa del.
- Uppgiften går ut på att skriva en utvärdering av uppgift 3.
- Skriv din utvärdering i textfilen `5_utvärdering_om_tid_kvar.txt` som finns i samma mapp som detta dokument.

Förslag på hjälpfrågor vad gäller utvärderingen:

- Finns det funktionaliteter som hade kunnat utvecklas för att göra programmet ännu bättre?
(Dessa funktionaliteter kan sträcka sig utanför uppgiftens beskrivning)
- Finns det bitar av din kod som eventuellt hade varit bättre att ha i en hjälpfunktion?
- Beskriv och motivera några val du gjorde när du arbetade med uppgiften.

Utvärderingen av **koden** är något som bedöms, samt ditt användande av **datavetenskapliga begrepp** när du kommunicerar kring koden.

Här är vad som står i bedömningsmatrisen vad gäller utvärdering och datavetenskapliga begrepp:

	C	A
Utvärdering	Du utvärderar med nyanserade omdömen hur arbetet har gått.	Du utvärderar med nyanserade omdömen hur arbetet har gått. Du ger även förslag på förbättringar.
Datavetenskapliga begrepp	I din kommunikation kring uppgiften använder du med viss säkerhet datavetenskapliga begrepp.	I din kommunikation kring uppgiften använder du med säkerhet datavetenskapliga begrepp.

Programmering 1 - Ruby Cheat Sheet - Version till slutprov VT20

Köra ruby-program (i kommandotolken cmd.exe)

Ange kommandot **ruby** följt av namnet på filen som innehåller källkoden:

```
C:\>ruby my_source_code.rb
```

Navigera mellan mappar med kommandot **cd**:

```
C:\>cd MyFolder      byt mapp till MyFolder
C:\MyFolder\>cd ..   backa till föregående
```

Interactive Ruby Shell (irb)

Testa syntax och kodsnuttar med **irb**:

```
C:\>irb
irb(main):001:0>puts "Hello World!"
```

Värden, variabler och datatyper

Ett **värde** är ett stycke information (data) t ex talet 42, bokstaven K, ordet banan. En **variabel** är ett namn som symboliserar ett värde i koden.

Värden delas in i olika **datatyper** som har olika egenskaper och tillåtna operationer. Det går t ex inte (logiskt) att jämföra heltal med text på ett meningsfullt sätt.

Datatyp	Benämning	Exempel
Integer	heltal, int	843
String	sträng, text	"squiggle"
Boolean	bool	true false
Float	flyt- / decimaltal	34.8
Array	lista, fält	[4,7,1,9,12]

Input från användare

Funktionen **gets** (get string) låter användaren mata in en sträng som avslutas med enter. **chomp** tar bort radslutstecknet i slutet på strängen:

```
name = gets.chomp
```

Med **to_i** omvandlas input till ett heltalsvärde:

```
value = gets.to_i
```

Indentering (formatera källkod)

Korrekt **indentering** (indrag av kod-block med tab eller blanksteg) gör källkoden mer lättläst och enklare att felsöka. VS Code och andra kod-editorer kan göra detta automatiskt.

Tilldelning (assignment), initiering

En variabel har ingen relevans i koden förrän den tilldelats ett värde. Efter **tilldelning** symboliserar variabeln sitt tilldelade värde:

```
name = "Berit"      # tilldelning
puts name           # name innehåller "Berit"
```

Den första tilldelningen av en variabel i koden kallas **initiering**. Vid nästa tilldelning av samma variabel kommer värdet att förändras (skrivas över). **Tilldelningsoperatoren** = ska inte förväxlas med jämförelseoperatoren ==

Namngivning

Variabelnamn skall skrivas med **små bokstäver** (gemener). Inled inte med siffra. Undvik specialtecken (åäö, etc) och stora bokstäver (versaler) som ger konstant variabel. Använd **snake_case**, undvik CamelCase (konstant var). Reserverade ord som while, end, return osv kan inte användas som variabelnamn.

Tydlig namngivning innebär att variabler och funktioner ges *namn som syftar till deras innehåll eller funktion* i koden. Ex:

```
first_name = "Gunnar"      # bra namngivning
l33t_h4xx0r = "Gunnar"    # dålig
```

Kommentarer i källkod

Inleds med **#** och kan skrivas *efter en kodrad*:

```
sum = a + b      # summera talen a och b
```

Om raden *inleds med #* är hela raden kommentar:

```
# sum = a + b    (den här koden utvärderas ej)
```

Utskrift på skärmen (console output)

```
puts "Hello"      # utskrift med radbrytning
print "Hello"     # utskrift utan radbrytning
```

Funktionen **p** skriver ut på ett format som visar utskriftens datatyp (bra vid debugging!):

```
p "Hello"
```

Vid utskrift kan variabler integreras i en given sträng genom **sträng-interpolering**:

```
age = 18
puts "You are #{age} years old"
```

If-satser (if-statements)

if-satser har ett eller flera kod-block som utförs om ett **villkor** är sant. Endast ett block utförs (även om flera villkor är sanna). Ex:

if-elsif-else:	if-else:
<pre>if x < y puts x elsif x >= y puts y else puts x + y end</pre>	<pre>if x > y puts x else puts y end</pre>

Operatorer för jämförelse (comparison)

Jämförelse måste ske mellan värden av samma datatyp. En jämförelse utvärderar alltid till **true** eller **false**.

x < y	x mindre än y
x > y	x större än y
x <= y	x mindre än eller lika med y
x >= y	x större än eller lika med y
x == y	x lika med y
x != y	x inte lika med y

Aritmetiska operatorer (numeriska värden)

a + b	addition
a - b	subtraktion
a * b	multiplikation
a / b	division
a % b	modulus (rest vid heltalsdivision)
a ** b	exponent ("a upphöjt i b")

Logiska operatorer

Används mest för att *kombinera flera villkor*. Använd gärna parenteser för komplexa villkor.

&&	logiskt "och", and fungerar likadant
	logiskt "eller", or fungerar likadant
!	logiskt "inte", not fungerar likadant

Typomvandling (type conversion)

Omvandla värden till andra datatyper:

25.to_s	# to string -> "25"
"42".to_i	# to integer -> 42
"1.23".to_f	# to float -> 1.23

Iteration med while-loop

En **while-loop** består av ett **villkor** och ett kod-block som upprepas (itereras) så länge villkoret är sant. Ex:

```
i = 0      # loop-varibel initieras
while i < 5 # kod-blocket upprepas fem ggr
  puts i
  i += 1   # inkrementera loop-varibel
end
```

I en **inkrementerande loop** (exemplet ovan) är villkoret beroende av ett ökande värde. I en **dekrementerande loop** är villkoret beroende av ett minskande värde, ex:

```
i = 5      # loop-varibel initieras
while i > 0 # kod-blocket upprepas fem ggr
  puts i
  i -= 1   # dekrementera loop-varibel
end
```

Nyckelordet **break** avbryter loopen direkt. Nyckelordet **next** påbörjar nästa iteration/varv.

Indexering

Enskilda tecken eller värden som ingår i en sträng eller lista/array kallas **element**. Varje element har ett unikt **index** (numrerad position) som ger åtkomst till elementet. *Index är noll-baserat* och ökar när man går åt höger. Ex:

element (tecken):	B	e	r	t	i	l
index:	0	1	2	3	4	5

Indexerings-operatorn [] skrivs efter variabeln eller värdet som indexeras. Index anges inom klammarna som värde eller variabel. Ex:

```
name = "Bertil"
puts name[4] # "i" skrivs ut
name[0] = "b" # ersätt element -> "bertil"
```

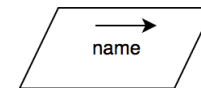
Speciella index-format:

name[-1]	# ger sista elementet -> "l"
name[-2]	# ger nästa sista osv -> "i"
name[1..3]	# intervall av element -> "ert"

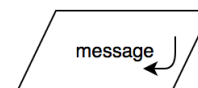
Flödesschema (flowchart)

En grafisk representation av **programflödet** med symboler för **sekvens**, **selektion** (if-sats) och **iteration** (loop). Dessutom används särskilda symboler för **input** och **output**.

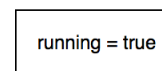
Input – en snedställd rektangel med en höger-riktad pil. Variabeln som tilldelas anges i symbolen.



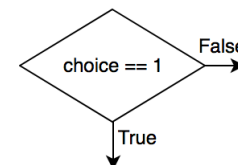
Output – en snedställd rektangel med en "retur"-pil. Variabeln eller värdet som matas ut anges i symbolen.



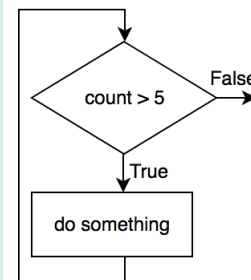
Sekvenssteg eller ovillkorligt steg – en vanlig rektangel. Operationen som utförs skrivs som kod eller pseudokod.



If-sats – en "diamant" med ett villkor i kod eller pseudokod. Symbolen har *vardera en utgång för true och false*.



Loop – en diamant med ett villkor där *flödet återvänder till villkoret* om det är sant. När villkoret inte uppfylls går flödet vidare.



Sträng-operationer (ett litet urval)

"Hello\n".chomp	# tar bort radslut -> "Hello"
"Hello".length	# ger längd -> 5
" Hello".strip	# tar bort space -> "Hello"
"one two".split	# ger lista -> ["one", "two"]
"Hello".upcase	# versaler -> "HELLO"
"Hello".downcase	# gemener -> "hello"

Egendefinerade funktioner (metoder)

En funktion kapslar in ett stycke kod med en tydlig och avgränsad uppgift. Input till en funktion kallas **argument** eller **parametrar** (noll eller flera). Output kallas **returvärde**. Funktionen måste vara *definierad innan den anropas*. Variabler inom funktionen är lokala (inte tillgängliga utanför funktionskroppen).

```
def sum_equal( a, b ) # a och b är argument
  sum = a + b
  if sum % 2 == 0
    return true      # returvärde
  else
    return false     # returvärde
end                  # slut funktionskropp

sum_equal( 2, 4 )    # funktionsanrop
```

Array (lista, fält)

En lista är en **datastruktur** som innehåller **element** av en viss datatyp, t ex heltalsvärden eller strängar. Listor har *många gemensamma egenskaper med strängar*, t ex **indexering**.

```
my_list = []      # initiera tom lista
# skapa lista av strängar:
numbers = ["one", "two", "three"]
numbers[1]        # indexering -> "two"
```

Lägg till element i slutet av listan genom att inkrementera listans högsta giltiga index:

```
numbers[3] = "four" # lägg till fjärde element
```

För att **ta bort element** kan man skapa en ny lista och kopiera över valda element, ex:

```
new_list = numbers[0..2]
```

Några begrepp

Kontrollstruktur – if-sats eller loop

Argument – input till funktioner och program

Input-loop – styrs av användarens input

Inkrementera – öka värde (vanligtvis med ett)

Dekrementera – minska värde

Syntaxfel – uppstår när språkets regler inte följs, t ex parentesfel eller "end" som saknas.

Logiska fel – koden går att köra men ger felaktiga resultat, t ex == förväxlas med =

Algoritm – recept eller standardlösning på ett givet problem.

Interaktivt program – interagerar med användaren genom meddelanden och input.

Filsystem

Några kommandon för att arbeta med filer i Ruby.

Dir.glob() Returnerar en array med strängar motsvarande fil- och mappnamn i nuvarande directory.

Exempel på argument till Dir.glob()

```
Dir.glob("*")  
Returnerar alla sökvägar
```

```
Dir.glob("*.html")  
Returnerar alla sökvägar som slutar med .html
```

File.directory?() Tar en sträng som argument och returnerar true om strängen är sökväg till en mapp utifrån nuvarande directory, annars returneras false.

File.file?() Som ovan fast returnerar true om argumentet är sökvägen till en fil.

File.readlines() Tar en sträng som argument som motsvarar sökvägen till en fil. Returnerar en array med varje rad från filen som varsitt element.

File.open() "Öppnar" en fil för att kunna läsa, skriva eller lägga till innehåll.

Exempel på argument till File.open()

```
File.open("sökväg", "w")  
Filen öppnas för att skrivas till, resulterar i en ny fil  
File.open("sökväg", "r")  
Filen öppnas för att läsas från  
File.open("sökväg", "a")  
Filen öppnas för att skrivas till, lägger till om befintlig fil (a: append)
```

Genom att tilldela en fil till en variabel går det enkelt att stänga filen med hjälp av **.close**

Exempel:

```
min_fil = File.open("textfil.txt", "a")
```

```
    # Redigerar filen
```

```
min_fil.close
```

Undantagshantering

Lite om undantagshantering i Ruby.

begin - rescue - end

Kodblock för att fånga upp och hantera undantag

Exempel:

```
def crash(x, y)  
  begin  
    z = x / y  
    rescue ZeroDivisionError => e  
      puts "Ett fel uppstod, division med 0."  
      puts "Ange ny nämnare och tryck ENTER."  
      y = gets.chomp.to_i  
      retry  
    end  
    return z  
  end  
end
```

Kommandot **retry** kör om koden i begin-blocket.

Om man istället vill avsluta programmet då undantaget uppstår kan kommandot **exit** användas. Exempel:

```
def crash(x, y)  
  begin  
    z = x / y  
    rescue ZeroDivisionError => e  
      puts "Ett fel uppstod, division med 0."  
      puts "Programmet avslutas"  
      exit  
    end  
    return z  
  end  
end
```

Flera undantag kan fångas upp inom samma begin - rescue - end - block. Förklarande skiss:

```
begin  
  # Kod som kan få ett undantag att uppstå  
  rescue ZeroDivisionError => e  
    # Hantering av undantaget ZeroDivisionError  
  rescue Errno::ENOENT => e  
    # Hantering av undantaget Errno::ENOENT  
end
```

Tips för att analysera vilka undantag som kan uppstå i en kod är att låta koden krascha och undersöka felmedelandet som ges, där står vilket undantag som fick koden att krascha.