Programmering 1 - Ruby Cheat Sheet Namn:

Interactive Ruby Shell (irb)

Testa syntax och kodsnuttar med **irb**:

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

Tips

Använd [tab] för att autofylla

Pil upp för att köra samma kod igen

[ctrl]+[c] för att stänga av oändlig loop

Läs felmeddelanden (framför allt vilken rad och titta även på raden över)

Skriv "ruby" innan dina körbara filer

Ha snabba och vettiga test-cases, hårdkoda in

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
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:

```
age = gets.to_i
```

Identering (formatera källkod)

Korrekt **indentering** (indrag av kod-block med tab eller blanksteg) gör källkoden mer lättläst och enklare att felsöka. VS Code och andra kodeditorer 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). **Tilldelningsoperatorn** = ska inte förväxlas med jämförelseoperatorn == .

Kommentarer i källkod

Inleds med # och kan skrivas efter en kodrad.

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

OBS! Om raden inleds med # är hela raden kommentar.

Namngivning

Variabelnamn skall skrivas med **små bokstäver**. Inled inte med siffra. Undvik specialtecken (ääö, etc) och stora bokstäver 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
```

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"
```

Logiska operatorer

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

&&	logiskt "och"
||	logiskt "eller"
!	logiskt "inte"

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")

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
"1.23".class       # klassen  => String
"1.23".to_f.class  # klassen  => Float
```

Iteration med while-loop

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

```
i = 0            # loop-variabel initieras
while i < 5      # kod-blocket upprepas fem ggr
   puts i
   i += 1        # inkrementera loop-variabel
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-variabel initieras
while i > 0      # kod-blocket upprepas fem ggr
   puts i
   i -= 1        # dekrementera loop-variabel
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 nollbaserat* 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 klamrarna som värde eller variabel. Ex:

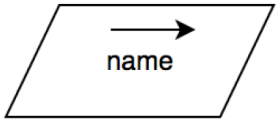
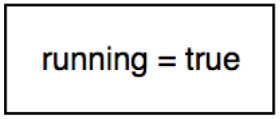
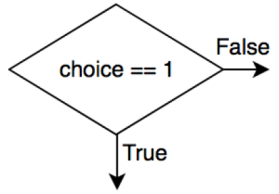
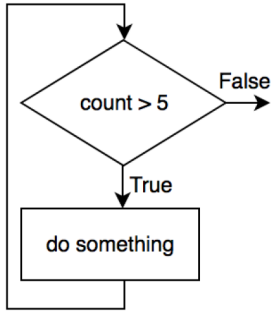
```
name = "Bertil"
puts name[4]     # "i" skrivs ut
name[0] = "b"    # tecknet på index 0 ersätts med b
```

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 högerriktad 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"
"a b".split         # ger lista        => ["a", "b"]
"Hello".upcase      # versaler         => "HELLO"
"Hello".downcase    # gemener          => "hello"
```

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"
```

```
numbers << "four"         # elementet "four" läggs
                          # till i slutet av listan
numbers.append("five")    # elementet "five" läggs
                          # till i slutet av listan
```

```
names = ["Katniss", "Peter", "Primrose"]
names.delete_at(1)        # ["Katniss", "Primrose"]

names.insert(1,"hej")     # ["Katniss", "hej"
"Primrose"]
```

```
numbers = [3, 2, 5, 0, 1, 3, 7]
numbers[1..3]                      # => [2, 5, 0]
```

Egendefinierade 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).

Generell funktionsstruktur

```
def funktionsnamn(arg1, arg2, arg3...)
  #deklarera variabler
  #ev kopiera argument (out-of-place eller inplace)
  #algoritm (ev return i algoritm)
  #return
end

#testkod
```

Exempel

```
def sum_even(a, b)        # a och b är parametrar
   sum = a + b
   return sum % 2 == 0    # returvärde
end                       # slut funktionskropp

sum_equal(2, 4)           # funktionsanrop (=> true)
```

Dokumentation av kod

Dokumentation av funktioner och program skrivs ovanför koden. Inom kursen jobbar vi efter följande mall för dokumentation.

```
# Beskrivning:          ...
# Argument 1:           ...
# Argument 2:           ...
#   etc
# Return:               ...
# Exempel:
  ...  ...  ...  ...
  ...  ...  ...  ...
  ...  ...  ...  ...
  ...  ...  ...  ...
# Datum:                ...
# Namn:                 ...
```

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

Konkatenera - lägga ihop två strängar

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

tidskomplexitet - anger på vilket sätt tiden det tar att köra koden ökar när längden på input ökar.

ex: **O(n) - linjär tidskomplexitet** - tiden går att ungefär anpassa likt linjen y = k*x + m där x är storleken på input.

Kodstruktur - övergripande

```
# importera kod (t.ex require_relative)

# deklarera (globala) variabler

# funktioner

# main-funktion

# testkod

# main()
```

Undantagshantering

Lite om undantagshantering i Ruby.

Framkalla ett undantag med kommandot **raise**:
Undantaget blir per default *RuntimeError*.

```
def last_element_is_zero(input_array)
   if input_array.class != Array
      raise "Fel sorts input"
   end
   return input_array[-1] == 0
end
```

Kodblocket **begin – rescue – end**

```
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
```

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
```

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

Filsystem

Några kommandon för att arbeta med filer i Ruby.

File.read()
Tar en sträng som argument som motsvarar sökvägen till en fil, returnerar en sträng med filens innehåll.

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

File.open()
Tar två strängar som argument, den ena är en sökväg till en fil, det andra är hur man vill interagera med filen. Man kan läsa, skriva eller lägga till innehåll.

```
File.open("sökväg", "w")   # Filen öppnas för att
                           # skrivas till, ger 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
```

```
min_fil.close              # "stänger" min_fil
```