Linnæus University

Grunderna i Python

Tobias Andersson Gidlund

Agenda

- Tre viktiga koncept
  - Problemlösning med dem
- Enkla program
  - Utskrift, inmatning, beräkningar
- Variabler

Tre viktiga koncept

- Programmering, i alla fall imperativ, bygger på tre grundläggande koncept:
  - Sekvens
  - Selektion
  - Iteration
- Det är också de tre begreppen som ligger till grund för algoritmer
  - Problemlösning med hjälp av datorer
- Det är grunden, men det behövs även lite till

Variabler

- Data som används behöver lagras på ett eller annat sätt
  - För att kunna användas i till exempel beräkningar
- I grund och botten så lagras data i minnet i datorn som ettor och nollor
- För att kunna nå värdena så används *variabler*, det vill säga ett ord som pekar på det minnesutrymme där data finns
  - En variabel är alltså ett *namngivet utrymme i primärminnet där data finns*
- För att veta hur många minnesceller (byte) som ska läsas för en viss data, så har variabeln en *datatyp*
  - Kan vara heltal, decimaltal, tecken, sträng eller liknande

Listor

- En variabel som kan hålla fler än ett värde kallas för en *lista*
- De flesta programmeringsspråk innehåller olika typer av listor
  - Sådana som har en fast storlek, eller som ändrar storlek i takt med antalet element med mera
- Det här är ett exempel på en *datastruktur*, vilket är ett viktigt begrepp inom programmering
  - En möjlighet att lagra flera värden på ett effektivt sätt
  - Effektivt är olika för olika typer av datastrukturer, en del är snabba på uppsökning, andra på insättning

Instruktioner

- Det som sedan utför något är *instruktioner*
- Det kan vara *uttryck* eller *satser*
- Uttryck är matematiska beräkningar som utförs på samma sätt som i just matematiken
- Satser är de instruktioner (som kan innehålla uttryck) som faktiskt *utför* något
  - Kan vara kommandon som `print()` för att skriva ut något

Ett första program

- Just kommandot `print()` används för att skriva ut det som kommer mellan parenteserna
  - Det kan vara en matematiska beräkning (ett uttryck), en teckensträng eller liknande
- Om det som ska skrivas ut är text, skrivs de inom enkla apostrofer ’så här’, beräkningar skrivs matematiskt

```
print('Meningen med livet, universum och allting:')
print(36 + 6)
```

- Körning:

```
Meningen med livet, universum och allting:
42
```

Nice to have

- Med det så har vi det som behövs för att kunna skriva ett program som löser en uppgift
- För att göra livet som programmerare lättare, så lägger språk till saker
  - Från andra paradigmer till olika typer av funktioner och bibliotek
- Det är en av de saker som gör att olika språk skiljer sig åt
- När det gäller Python så kommer vi att titta på *funktioner* och *objektorientering*

Sekvens, variabler och instruktioner

Sekvens

- I ett datorprogram så utförs saker en i taget och efter varandra
  - Sant så länge man inte har flertrådade program, men det ligger utanför kursen
- I följande pythonprogram ska alltså de tre utskrifterna komma efter varandra (på en egen rad)

```
print("Hello")
print("World")
print("!")
```

- Körning:

```
Hello
World
!
```

`print()` instruktionen

- I Python är `print()` en kraftfull och viktig instruktion
  - Det är en *sats*
  - Den är också en *funktion*
- Python har många inbyggda funktioner som är tillgängliga från början i språket
  - Omkring 150 olika inbyggda funktioner
- De har alla samma grundläggande struktur:
  - Ett namn som startar med en liten bokstav
  - Parenteser runt det som ska skickas till instruktionen (funktionen)

Variabler

- Som tidigare sagts så används variabler för att lagra data
- Alla variabler har en datatyp och ett namn
  - Ett namn som börjar på en bokstav och sedan siffror eller understykningsstreck
  - Namnet används sedan för att komma åt värdet (eller ändra det)
- Variablen *tilldelas* ett värde med likamedtecken (=) (vi kallar det tilldelningstecken)
- I Python behöver vi inte uttryckligen tala om vilken datatyp en variabel har utan det bestäms vid tilldelning

Mer om variabler och datatyper

- Även om det inte är nödvändigt att ange en datatyp vid deklaration så har alla variabler en datatyp
  - Datatyperna är heltal, flyttal, strängar, olika klasser med mera (mer senare)
- Med den inbygda funktionen `type()` är det möjligt att fråga en variabler vad den har för datatyp

```
a = 42
print(type(a))
a = 3.14159
print(type(a))
```

- Utskrift

```
<class 'int'>
<class 'float'>
```

Exempel

- Exempelkod:

```
a = 5                   # Variabeln är ett heltal
mess = 'Star Wars'      # Variabeln är en sträng

print(a)
print(mess, 'rules!') # Vi slår ihop två strängar
a = a + 5               # Vi lägger till fem till det som redan finns i a
print(a)
```

- Körning:

```
5
Star Wars rules!
10
```

Beräkningar

- Exempelkod:

```
num1 = 36
num2 = 6

print(num1, '+', num2, '=', num1 + num2)
print(num1 - num2)
print(2 * (num1 + 5))
print(num1 % num2) # Restoperator (det som blir över)
print(39 / 18)
```

- Körning:

```
36 + 6 = 42
30
82
0
2.1666666666666665
```

Programmera med stil

- Det finns många regler för hur kod ska skrivas för att fungera, men det finns även *riktlinjer* för hur den ska se ut
- De här riktlinjerna är inte något som programmeringsspråket hanterar i sig, utan de är i stort frivilliga
- För att kunna läsa kod snabbt är det dock en mycket bra idé att följa någon form av riktlinje
  - För Python är de i stort definierade i PEP-8
- En sådan regel är att ha mellanrum mellan tal och operatorer:

```
print(5 + 2 / 6)
```

- Programmet skulle fungera precis lika bra utan:

```
print(5+2/6)
```

F-strängar

- Ofta vill man *formatera* sin utskrift så att den ser rätt ut
  - Hur många decimaler som ska skrivas ut eller inledande nollor
- I Python kan man göra det med *f-strängar* där man sätter ett *f* före strängen
- I texten där man vill skriva ut någon variabel sätter man en *platshållare* med en *formatspecifikation*

```
print(f'{variabelnamn:hur_formateras}')
```

Exempel

- Skriva ut 39/17 med två decimaler:

```
num = 39 / 18
print(f'Det blir {num:.2f}') # Notera att värdet avrundas korrekt
```

- Körning:

```
Det blir 2.17
```

- Skriva ut med inledande nollor:

```
pi = 3.14169
print(f'Med lite utrymme: {pi:010.3f}') # Inled med 0, ger 10 tecken totalt
```

- Körning:

```
Med lite utrymme: 000003.142
```

Mer matematik

- Det är ganska vanligt att man behöver göra beräkningar av olika slag i ett datorprogram
- Förutom de fyra räknesätten samt restoperatorn, har Python tillgång till ett *standardbibliotek* för matematik
  - Det heter `math` och behöver läsas in högst upp i programmet med `import math`
- Med det ges tillgång till bra värden på *pi* och *e*, men också sätt att räkna upphöjt, beräkna logaritmer eller kvadratrötter och mycket mer
- För att använda dem skriver man `math.` och namnet, t.ex. `sqrt()` och skickar med parametrar

Exempel på matematik

- Exempel:

```
import math

sida1 = 4
sida2 = 3

res = math.sqrt(sida1 ** 2 + sida2 ** 2)
print(f'Hypotenusans längd: {res:.2f}')
```

- Körning:

```
Hypotenusans längd: 5.00
```

Ytterligare ett bibliotek

- Något man ofta behöver göra är är att skapa (mer eller mindre) slumpmässiga tal
- Det gör man genom att använda biblioteket `random`
- Här finns det möjlighet att skapa både enstaka slumptal eller listor (mer senare) av slumptal
- Den vanligaste funktionen är `randint()` som tar intervallet som parametrar
  - Båda talen är inkluderade i intervallet

Exempel på tärningsslag

```
import random

t = random.randint(1, 6)
print(t)
```

- Körnig

```
3
```

- Slå två tärningar:

```
import random

t1 = random.randint(1, 6)
t2 = random.randint(1, 6)
print(t1 + t2)
```

- Körning:

```
8
```

Fel

- Ja, när man programmerar så blir det fel ibland...
  - Ganska ofta, om vi ska vara ärliga
- Programmering är ganska ofta en cykel av att skriva kod → köra/krascha → läsa felmeddelanden → skriva kod
- En viktig del av att vara programmerare är att kunna läsa och förstå de fel som uppstår
  - Kvaliteten på felmeddelanden är något som skiljer olika programmeringsspråk åt
  - Rust brukar anses ha mycket bra felmeddelanden, medan C är känt för att ha de sämsta

Pythonfel

- Python anses ha ganska bra felmeddelanden
  - Tillräckligt uttryckliga för att förstå och pekar ofta på rätt ställe i koden
- Notera att felmeddelanden inte ges för alla typer av fel
- Det finns tre olika kategorier av fel:
  - Syntaxfel
  - Exekveringsfel
  - Logiska fel (utan felmeddelanden)

Syntaxfel

- När språkets regler inte följs kommer Python att ge ett *syntaxfel*

```
print('Hello world'
```

- När koden ovan körs, får man följande felmeddelanden:

```
File "/home/tanmsi/tmp/pyrk/test.py", line 1
  print('Hello World'
        ^
SyntaxError: '(' was never closed
```

- Den här klassen av fel är relativt enkel att förstå
- I många fall hjälper en utvecklingsmiljö också till att se den här typer av fel innan kompilering

Exekveringsfel

- Den här typen av fel uppstår när koden körs
  - Kan till exempel bero på att användaren har skrivit in fel typ av data
- Det viktiga att komma ihåg är att syntaxen är rätt vilket innebär att felet inte kan hanteras under själva kompileringen
- Ett klassiskt exempel är division med noll

```
print(42 / 0)
```

- Vid körning

```
Traceback (most recent call last):
  File "/home/tanmsi/tmp/pyrk/test.py", line 1, in <module>
    print(42/0)
          ~~^~
ZeroDivisionError: division by zero
```

Logiska fel

- Det mest svårfångade problemet eftersom det innebär att man har tänkt fel 😇
- Det här felet syns bara när man kör och när koden testas
  - Vilket är anledningen till varför testning är så viktigt
- En rekommendation är att inte försöka lösa alla delar av ett problem på en gång, utan hellre dela upp det i mindre delar först
- Det är också viktigt att köra sin kod ofta och se vad som händer

Selektion och logiska uttryck

Selektion

- Ofta behöver man välja mellan olika sekvenser att utföra i sitt program och då används selektion
- Rent programmeringsmässigt kan det göras på flera sätt, `if` är dock det vanligaste

```
if 3 > 4:
  print("Sant")
else:
  print("Falskt")
```

- Körning:

```
Falskt
```

Logiska uttryck

- Det intressanta här är vad man gör en jämförelse mellan
  - I exemplet om 3 är *större än* fyra
- Det som utvärderas är ett *uttryck* och dem *måste* kunna utvärderas till antingen `sant` eller `falskt`
- Vi kan alltså *inte* fråga ”höger eller vänster?”
- Frågan måste göras om till något som kan utvärderas till sant eller falskt
  - Till exempel: ”ska vi gå åt höger?”

Mer om selektion i Python

- Varje rad med en selektion avslutas med ett kolon
- Alla rader under en selektion måste vara indragna
- Man kan använda bara `if` om inga alternativ finns
- Har man bara ett annat alternativ används `else` i stället för `elif`
- Observera att endast *ett* av alla de olika alternativen utförs, det som *först* utvärderas till sant

```
if logiskt_uttryck:
  satser
elif logiskt_uttryck:
  satser
elif logiskt_uttryck:
  satser
else:
  satser
```

Exempel

- Exempel:

```
namn = 'Anakin'

if namn == 'Anakin':
  print('Darth Vader!')
```

- Körning:

```
Darth Vader!
```

- Exempel:

```
namn = 'Pink Floyd'

if namn == 'Anakin':
  print('Darth Vader')
else:
  print('Annat namn')
```

- Körning:

```
Annat namn
```

Ytterligare exempel

- Exempel:

```
a = 42

if a > 42:
  print('Stort')
elif a < 42:
  print('Litet')
else:
  print('Exakt')
```

- Körning:

```
Exakt
```

Logiska uttryck med logiska operatorer

- Det logiska uttrycket kan också bestå av flera logiska uttryck som sätts samman
  - Med de logiska operatorerna *och*, *eller* samt *icke*
  - I Python smart nog kallade *and*, *or* samt *not*
- Två uttryck med *and* mellan sig är sant om *båda* uttrycken är sanna
- Med *or* mellan så måste minst ett vara sant
- *not* är sant om det efter är falskt

Exempel

- Exempel:

```
temp = 22
nederbord = False

if temp >= 20 and not nederbord:
  print('Soligt och tort!')
elif temp >= 20 and nederbord:
  print('Varmt och blött')
else:
  print('Osäkert väder')
```

- Körning:

```
Soligt och tort!
```

Nästlade selektioner

- En selektion kan ha ett antal satser, inte bara en rad, och eftersom `if` är en sats, så det finnas flera sådana i en
  - Det kallas för att *nästla* satserna
- Exemplet tidigare kan skrivas om så att vi först kontrollerar temperaturen och sedan nederbörden
  - Fördelen då är att man *enbart* kontrollerar nederbörden om temperaturen är rätt
- Det är här viktigt med indenteringen för att kunna hålla reda på vilken `if` en sats tillhör

Exempel

- Exempel:

```
temp = 22
nederbord = False

if temp >= 20:
  if not nederbord:
    print('Soligt och tort!')
  else:
    print('Varmt och blött')
else:
  print('Osäkert väder')
```

- Körning:

```
Soligt och tort!
```

Iteration

Iteration

- Om en sekvens behöver upprepas används någon form av iteration
- I Python finns flera olika sätt att iterera, vanligast är `while` och `for`

```
a = 1

while a < 3:
  print(a)
  a = a + 1
```

- Körning:

```
1
2
```

Upprepa med `while`

- Om man på förhand inte vet antalet iterationer som ska göras så används `while`
- Den tar ett *logiskt uttryck*, precis som selektionen, och utför ett antal satser *så länge uttrycket är sant*
- Det är därför viktigt att ändra någon parameter i det logiska uttrycket
  - Annars är det lätt att hamna i en evighetsloop
- Precis som för selektion måste satserna som tillhör en iteration indenteras

Inledande exempel

- Det här exemplet utgår ifrån att det finns 100 pengar i variabeln `money`
- Så länge det är möjligt att ta bort 10 pengar, så kommer `while`-satsen att göra det

```
money = 100
print(f'Money: {money}')

while money > 0:
    money = money - 10

print(f'Money left: {money}')
```

- Körning:

```
Money: 100
Money left: 0
```

Exempel (från boken)

- I det här exemplet ska de jämna talen mellan 0 och 6 (ej inkluderat) skrivas ut mellan två hakparenteser

```
print('[', end = '') # Ingen ny rad läggs till
k = 0

while k < 6:
  print(f'{k:2}', end = '')
  k = k + 2

print(']')
```

- Körning:

```
[ 0 2 4]
```

Mer om iteration

- Det kan givetvis finnas flera satser, även selektion och iteration, i en iteration
  - Nästlade iterationer tas upp senare
- Återigen, kom ihåg att indentera korrekt för att veta vad som tillhör vilken sats
- Undvik att använda `while True` utan försök hitta ett lämpligt villkor
  - En `while` som alltid körs kan avbrytas med `break`
  - Kan också användas i program där man behöver avbryta hastigt

Exempel

- Följande exempel lägger ihop alla jämna tal mellan 0 och 100

```
n = 0
sum = 0

while n < 100:
  if n % 2 == 0:     # Om talet är jämnt
    sum += n         # Lägg till n till sum
  n += 1             # Öka n med ett

print('Summan blir', sum)
```

- Körning:

```
Summan blir 2450
```

Räkna tärningar

- Det här exemplet kastar en sexsidig tärning och beräknar hur många gånger det behöver göras för att få en summa över 20

```
import random

sum = 0
rolls = 0

while sum <= 20:
    sum += random.randint(1, 6)
    rolls += 1

print(f'It took {rolls} rolls to get 20.')
```

- Utskrift:

```
It took 6 rolls to get 20.
```

Exekvera i evighet

- Det är enkelt att hamna i en evighetsloop
  - Kanske glömmer man att uppdatera en variabel
  - Fel logik används
- Det händer alla!
- Olika miljöer har olika sätt att avsluta en evighetsloop
  - I terminalen trycker man `Ctrl + C`
  - I Jupyter, och de flesta IDE:er, finns det ofta en ”stoppknapp” som man kan trycka på

Iteration med `for`

- Om man vet antalet iterationer som behöver göras kan man använda `for`
  - Det vill säga om värdena är inom ett *intervall*
- Den generella formen är:

```
for räknare in range(startvärde, slutvärde, steg)
```

- `steg` kan uteslutas om man vill gå ett steg (dvs om man vill gå 1, 2, 3... eller liknande)
- Variabeln kallad `räknare` får värdet för varje steg i iterationen
- Observera att `slutvärde` ska vara första värdet *utanför* intervallet
  - Det kommer alltså aldrig att nås

Ett första exempel

- Ett enkelt, första, exempel är att skriva ut alla siffror i ett intervall:

```
for i in range(10):
    print(i, end='')
```

- Körning:

```
0123456789
```

- Notera hur intervallet, skrivet så här, går från 0 till 9
- Vill man i stället gå från 1 till 10, får man skriva:

```
for i in range(1, 11):
    print(i, end='')
```

- Körning:

```
12345678910
```

Exempel

- Samma exempel som för `while` men nu med `for`

```
print('[', end = '')

for k in range(0, 6, 2): # Från 0 till 6 (ej inkluderad)
  print(f'{k:2}', end = '')
print(']')
```

- Körning:

```
[ 0 2 4]
```

Gå baklänges

- Det är också möjligt att ange ett negativt steg (och gå baklänges)

```
for i in range(10, 0, -1):
  print(i, '', end='')
```

- Körning:

```
10 9 8 7 6 5 4 3 2 1
```

- Notera att intervallet då också måste vara anget som ovan (startvärdet högre än slutvärdet)
- I exemplet går vi bara ett steg i taget, men det går att ange fler steg för varje iteration

Iterationer och flyttal

- En viktig sak att känna till om iterationer är att de behöver använda diskreta värden (som heltal, listor eller strängar (mer senare))
- Den interna representationen av decimala tal är inte exakt
  - Det är för att det skulle ta för mycket utrymme och kraft i relation till vad det ger
- Oftast så är den precision man får med flyttal tillräcklig, men det blir problem när de används tillsammans med iteration

```
i = 1

while i < 10:
    i = i + 0.1
    print(i)
```

Körning

- Flera rader har tagits bort för att hålla nere längden

```
1.1
1.2000000000000002
1.3000000000000003
1.4000000000000004
1.5000000000000004
1.6000000000000005
1.7000000000000006
1.8000000000000007
1.9000000000000008
2.000000000000001
2.100000000000001
2.200000000000001
2.300000000000001
2.4000000000000012
2.5000000000000013
2.6000000000000014
2.7000000000000015
2.8000000000000016
2.9000000000000017
3.0000000000000018
3.100000000000002
3.200000000000002
3.300000000000002
3.400000000000002
3.500000000000002
```

```
7.6999999999999895
7.799999999999989
7.899999999999989
7.9999999999999885
8.099999999999989
8.199999999999989
8.299999999999988
8.399999999999988
8.499999999999988
8.599999999999987
8.699999999999987
8.799999999999986
8.899999999999986
8.999999999999986
9.099999999999985
9.199999999999985
9.299999999999985
9.399999999999984
9.499999999999984
9.599999999999984
9.699999999999983
9.799999999999983
9.899999999999983
9.999999999999982
10.099999999999982
```

Nästlad iteration

- Konceptet med nästlad *selektion* (en `if` i en `if`) var inte så konstigt
- Nästlad iteration är inte heller det, men ofta något mer svårt att förstå
- Det är egentligen ganska enkelt, för *varje* steg i den yttre loopen ska *alla* steg i den inre utföras

```
for i in range(1, 3):
  for j in range(1, 6):
    print(i, '*', j, '=', i * j)
```

Körning

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5

2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
```

Ytterligare exempel

```python
# Antal rader i pyramiden
rader = 5

# Skriv ut varje rad i pyramiden
for i in range(1, rader + 1):

  # Skriv ut mellanslag
  for j in range(rader - i):
    print(" ", end="")

  # Skriv ut asterisker
  for k in range(i * 2 - 1):
    print("*", end="")

  print()
```

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

Sammanfattning av tre grundstrukturer

- Med det har vi diskuterat de tre grundstrukturerna i programmering
  - Sekvens
  - Selektion
  - Iteration
- Dessutom har vi tittat på variabler
- Det är dock viktigt att veta att det är ganska mycket kvar att lära om det ovanstående
- Läs gärna boken för att få mer kött på benen!