UPPSALA UNIVERSITET

OU4, exempeltenta, ”allt”

Carl Nettelblad

2020-03-03

Info

- Gå igenom rapporterade framsteg
- Har du rester du har missat?
- Har du redovisat och vi har tappat bort det?
- Jättebra om vi kan reda ut det den här veckan

UPPSALA UNIVERSITET

OU4

- Ibland krångel att få paket att fungera
  - Python är lättare än många andra språk/miljöer
  - Uppgiften tillrättalagd jämfört med ”verkligheten”
- Filer igen
  - Läsa filer tar tid
  - Läs in *en* gång med `load_csv`
  - Spara det lexikonet
- Arbeta rad för rad
  - Inte först bygga upp lexikon med värdena som strängar och sedan göra till flyttal

UPPSALA UNIVERSITET

Gör en gång

- Vi har länder och vilka färger vi vill ha för de länderna
- För varje land är det lätt att plocka ur lexikonet
  - Hämta ur lexikon görs enklast med `[ ]`
    - `get`-metoden fungerar, men ”ovanlig” för att bara hämta normalt
- Onödigt att plocka ut i förväg och lägga i separat lexikon/lista

- Hur gör vi för att loopa över färger och länder samtidigt?

UPPSALA UNIVERSITET

Alternativ 1

```
countries = ['fin', 'swe']
colors = ['blue', 'red']

i = 0
for country in countries:
    ax.plot(time, lex[country], ':'
            color=colors[i])
    …
    i += 1
```

UPPSALA UNIVERSITET

Men…

- Vi vill inte räkna index själva

UPPSALA UNIVERSITET

Alternativ 2

```
countries = ['fin', 'swe']
colors = ['blue', 'red']

for i, country in enumerate(countries):
    ax.plot(time, lex[country], ':’,
            color=colors[i])
    …
```

UPPSALA UNIVERSITET

Men…

- Behöver vi egentligen indexet? Vi vill ha en färg, inte ett tal.
  - Använder bara talet för att plocka fram färgen.

UPPSALA UNIVERSITET

Alternativ 3

```
countries = ['fin', 'swe']
colors = ['blue', 'red']

for country, color in zip(countries, colors):
    ax.plot(time, lex[country], ':',
            color=color)
    …
```

UPPSALA UNIVERSITET

Men…

- Behöver vi verkligen två listor?
  - Kanske bra, kanske dåligt
  - Beror på hur nära vi tycker att färger och länder hör ihop
- Två alternativ
  - En lista med tupler
  - Ett lexikon med landskoder och färger

UPPSALA UNIVERSITET

Alternativ 4

```
countries = [('fin', 'blue'), ('swe', 'red')]

for country, color in countries:
    ax.plot(time, lex[country], ':',
            color=color)
    …
```

UPPSALA UNIVERSITET

Alternativ 5

```
countries = {'fin': 'blue', 'swe': 'red'}

for country, color in countries.items():
    ax.plot(time, lex[country], ':',
            color=color)
    …
```

UPPSALA UNIVERSITET

Alternativ 6

```
countries = {'fin': 'blue', 'swe': 'red'}

for country in countries:
    ax.plot(time, lex[country], ':',
            color=countries[country])
    …
```

UPPSALA UNIVERSITET

Vad är ”bäst”?

- Jag tänker på det som att vi har länder och också vill ha vettiga färger. Det talar för `zip` av två listor.
  - Eller så använder vi de färdiga färgerna från `matplotlib`...

UPPSALA UNIVERSITET

Alternativ 7

```
countries = ['fin', 'swe']

for i, country in enumerate(countries):
    ax.plot(time, lex[country], ':',
            color=f'C{i}')
    …
```

Legend och label

- Enklaste sättet att se till att bara vissa kurvor syns i teckenförklaringen är att ge parametern label till bara vissa plottar.

```
ax.plot(time, smooth_a(lex[country], 5), ':',
        color=f'C{i}', label=country)
```

Annan variant:

```
curve, = ax.plot(time,
                 smooth_a(lex[country], 5),
                 ':', color=f'C{i}')
curve.set_label(country)
```

UPPSALA UNIVERSITET

Returvärde från `matplotlib`

- Många av plotfunktionerna returnerar objekt som representerar det man har ritat
- Kan användas för att ändra i efterhand
  - Här, lägga på label
  - Byta data i redan gjord plot, lägga på markörer, animera, …

UPPSALA UNIVERSITET

`intersection`

- Läs vad en funktion ska ta in och vad den ska returnera
  - Returnera betyder *inte* print
    - En funktion ska inte skriva någonting alls om det inte direkt ingår i dess syfte
    - Du vet inte hur den som *anropar* funktionen vill använda den
  - Om det står att funktionen ska ta in listor ska den ta in listor
    - Listor har ingen `keys`-metod, till exempel
    - Om den fungerar på något annat än listor *också* är det okej

UPPSALA UNIVERSITET

intersection

```
def intersection(l1, l2):
    return [i for i in l1 if i in l2]
```

UPPSALA UNIVERSITET

Om `l2` är en lista…

- `in` för en lista måste i värsta fall söka genom hela listan
  - Hitta att elementet finns/konstatera säkert att det inte finns
- `set` är i teorin bättre
  - Skapa ett index över listvärdena och kolla snabbt om de finns

```
def intersection(l1, l2):
    s2 = set(l2)
    return [i for i in l1 if i in s2]
```

Har man `set(a)` får man `set(b)`

```
def intersection(l1, l2):
    return set(l1).intersection(set(l2))
```

- Att ta snittet mellan mängder är ju något ganska naturligt, så det finns det en färdig metod för.
- Finns ”snyggare” syntax:

```
def intersection(l1, l2):
    return set(l1) & set(l2)
```

UPPSALA UNIVERSITET

Kommentarer

- Om vi verkligen vill returnera en lista får vi göra `list(…)` på vår resultatset
- & och and gör olika saker
  - and samordnar logiska uttryck
  - Om de första är ”sanna” i någon mening blir resultatet det sista uttrycket
  - & betyder ”kombinera de här värdena”
- Var *mycket försiktiga* med att använda and för något annat än hela logiska uttryck
- Lätt att tro att `1 <= x and y <= 5` betyder `1 <= x <= 5 and 1 <= y <= 5`

UPPSALA UNIVERSITET

OU5

- Allt gammalt är nytt – tillbaka till OU1
  - Vad får jag om jag skriver:

```
a = turtle.Turtle
a = turtle.Turtle()
a.xcor
a.xcor()
```

- Det är skillnad på ett uttryck som beskriver en klass, en funktion eller en metod och ett uttryck som skapar en instans av en klass eller *anropar* funktionen eller metoden
  - Vi säger ”anropa”, inte ”ropa på”, ”åkalla” eller ”kalla [på]”

UPPSALA UNIVERSITET

Svårt att komma igång?

- Titta noga på exemplen från lektion 8 och föreläsningen om egna objekt (föreläsning 5)
- Skriv klart `Vehicle` och `Light`
- Skriv klart `Lane`
  - Läs separata specifikationssidorna för de klasserna
  - Kan ge mer handfast ingång
- Kolla vad demofunktionerna för `Lane` och `Light` egentligen gör
  - De visar hur man skapar objekt av de båda klasserna, hur man anropar deras metoder o.s.v.

UPPSALA UNIVERSITET

Tenta

- Övningstenta från i höstas
- Finns två andra i Python med lösningsförslag

- Tänk på vad *vi* pratat om i kursen, men upplägg är i grunden detsamma

- Vi har pratat om att skriva och läsa dokumentation också

Struktur

- A-del
  - Flersvarsdel
    - Ringa in ett svar per alternativ, om inte annat anges
  - Kortsvarsdel
    - Fyll i svaret i rutan, på separat blad om du absolut inte får plats
- 75 % ska vara i huvudsak rätt för betyg 3
  - Underkänd A-del leder till att B-del inte rättas

UPPSALA UNIVERSITET

Vad vill vi se?

- Du kan läsa kod.
- Du kan skriva (begriplig) kod.
- Du kan använda syntaxen och standardfunktionerna i Python på ett rimligt sätt.
- Du kan använda strängar, listor och lexikon.
- Du förstår hur variabler, primitiva värden och referenser hanteras.
- Du vet vad modul, klass, objekt, variabel, parameter, funktion, metod är för något.
  - Vi har inte haft samma fokus på distinktionen argument/parameter.

UPPSALA UNIVERSITET

Kommentarer A1

A. Vi sorterar efter hela tupeln, alltså på talen först, i stigande ordning. Sedan plockar vi ut värdena i en listbyggare och skriver ut dem.

B. Lokala variabler och parametrar i funktioner påverkar inte variabler utanför. (Däremot påverkas de objekt de refererar till, men här var det primitiva värden i form av int.)

C. `range` tar en stegparameter, är högerexklusiv. Info om detta fanns i referensbladet.

D. Uttryck har typ i Python. Variabler och parametrar har i sig ingen fast typ, utan det beror vilket uttryck de tilldelas. Frågan kan tolkas som vilken typ `x` får i just detta anrop och borde kanske undvikas. Om raden `y = f(2)` saknats hade den tolkningen inte varit möjlig.

E. Vi har oftast sagt initierare, men konstruktor förekommer också.

UPPSALA UNIVERSITET

Kommentarer A1

F. Parametrar. Värdena som parametrar får i ett specifikt anrop kan kallas argument.

G. Övning på att listor kan användas med addition och multiplikation (upprepad addition). Parenteserna ej strikt nödvändiga, kan inte innebära tupel eftersom inget komma förekommer.

H. Vi kan inte säga något säkert. Många klassers namn börjar på stor bokstav, men det går att skapa funktioner eller andra identifierare som också börjar med stor bokstav.

UPPSALA UNIVERSITET

Kommentar A2

- I och med att kravet är positiva heltal är det ganska enkelt.
- Man kan mena att `z` borde vara `1` och inte `1.0`.
- Man skulle kunna skriva _ i stället för `i` när indexet inte används.
- Man skulle kunna skriva `z *= x`
- Man bör inte skriva `range(1, y + 1)` i stället
  - Det ”naturliga” sättet att räkna y gånger i Python är `range(y)`, skriv bara något annat om du faktiskt använder indexet

UPPSALA UNIVERSITET

Kommentar A3

- Viktigt att ursprungslistan inte förändras om man inte aktivt säger att det ska göras.
- Alltså inte okej att anropa `lst.sort()`
  - Däremot okej med

    ```
    lst = lst.copy()
    lst.sort()
    ```

  - Men `sorted` gör samma sak snäppet bättre
- Kan skrivas som enradare `return sorted(lst)[k]`

UPPSALA UNIVERSITET

Kommentar A4

- Eftersom vi vill gå igenom en lista och samtidigt ha index är `enumerate` ett naturligt val
- Eftersom vi bygger en lista element för element utan att titta på andra element är listbyggare ett naturligt val

UPPSALA UNIVERSITET

Kommentar A5

- Glöm inte bort att skriva själva klassdeklarationen
- Glöm inte bort `self`-parametern till metoder och att referera till fälten med `self`
- Kom ihåg att man *ofta*, men inte *alltid* lagrar parametrarna till initieringsmetoden som fält med samma namn (`Lane` i OU5 gör det *inte*)
  - Andra uppgifter kan säga vilka fält som ska/inte ska finnas

UPPSALA UNIVERSITET

Kommentar A6

- Väldigt snarlik vad vi gjorde i OU3
- Eftersom vi inte skapar en post i lexikonet per post i ursprungslistan är det här *inte* ett fall för lexikonbyggare
- Själva tilldelningen kan också göras utan if-satsen med till exempel

```
res[w] = res.get(w, 0) + 1
```

- Här utnyttjar vi aktivt att `get` kan ge ett standardvärde när nyckeln saknas
  - I fallet då nyckeln ej finns får vi `0`, adderar `1` och lägger det i lexikonet
- `split` fanns nämnd i referensbladet

Kommentar A7

- Kan vara enklare att använda randint
  - Kom ihåg att randint är inklusive slutpunkten
  - Så `random.randint(0, len(lst) - 1)`
  - Finns även random.randrange som fungerar som range `random.randrange(len(lst))`
  - Så potentiellt
    `return lst[random.randrange(len(lst))]`
- Allmänt: Tänk på skillnaden mellan att returnera ett värde och indexet för ett värde
- Lösningsförslaget saknar andra halvan av uppgiften

```
print(random_value(['a', 'b', 'c']))
```

UPPSALA UNIVERSITET

Kommentar A8

- Översätt den matematiska formeln till Python-uttryck
- Olika notation x, y och a, b (`a[0] == x1`, `a[1] == y1`)
- Har du glömt bort hur man skriver upphöjt? Gör bara upprepad multiplikation för att få kvadraten.

- Går att göra saker med loopar, `sum`, listbyggare, funktionen `reduce` (som vi inte pratat om), men uppgiften frågade inte om allmänna vektorer, bara par

UPPSALA UNIVERSITET

Del B

- Läsa kod. Börja med att titta på vad som finns givet. Går det att förstå koden?
- Är det våra lösningar som ska anropa den givna koden, eller är det den givna koden som anropar vår? Eller både och?
- Här *beskrivs* bara vad en av metoderna gör utan att vi får hela koden.

UPPSALA UNIVERSITET

Kommentar B1

- Här lagrar vi aldrig `n` i objektet, utan vi lagrar en nästlad lista som ett fält
  - Står inte i uppgiften vad fältet ska heta, här valdes `b`
  - Men det står att det ska vara det enda attributet
    - Storleken framgår sedan av `len(self.b)` ändå
  - Lösningsförslaget gör *fel* i att skriva `self.sz = sz`
- Lösningsförslaget har två nivåer av loopar, går förstås att skriva båda som listbyggare
- Hela klassen liknar `Lane` lite grann, men den är tvådimensionell

UPPSALA UNIVERSITET

Kommentar B2

- Här explicit loop med index
  - Kunde också skrivas som

```
for row in self.b:
    for cell in row:
        if cell != '.':
            c += 1
```

- Inte uppenbar kandidat för listbyggare
  - Vi vill ju summera över (vissa av) elementen

UPPSALA UNIVERSITET

Kommentar B3

- Vi har inte fokuserat på felhantering med `raise`
- Däremot har vi tagit upp `assert`
- I grunden en ganska enkel kontroll och uppdatering av innehållet i vår datastruktur
  - Liknar `Lane.enter`, men med felkontroll

UPPSALA UNIVERSITET

Kommentar B4

- Vi kan bygga upp en lista och sedan sätta ihop den till en sträng med `join` och tomma strängen som avskiljare
- Vi kan också bygga upp en lista per rad, sätta ihop den med `''.join()` och sedan göra `'\n'.join` på en lista med de strängarna för att bygga en sträng för hela spelplanen
  - Se lösningsförslag på en av de andra tentorna
- `__str__` ska absolut *inte* göra `print`
  - Den kan däremot (förstås) använda formatsträngar `f''`

UPPSALA UNIVERSITET

Kommentar B5

- Läs noga. Jämför bilden i B4 med exempel på listan från `get_configurations(self)`
  - Denna metod listar alla vågräta rader, alla lodrätar kolonner, de båda diagonalerna
  - Villkoret att någon har vunnit är alltså att något av dessa element består av enbart X eller enbart O
  - Ett sätt att kolla blir att ta fram de båda mallsträngarna
    - Enklare att bara göra `'X' * len(self.b)` för att göra det
  - Kan också använda `i.count('X') == len(self.b)` och se om den är `len(self.b)`
  - Eller `config.count(x_win)`

UPPSALA UNIVERSITET

Python

- *Uttryck* och *satser*
- Varje uttryck har en *typ*
  - `int, float, tuple, bool, list, str, dict, set…`
  - `Turtle, Rectangle, Lane, Light, reader`
- Exempel på uttryck:

```
3, ’hej’, True, a, b, len(a), len(b), a + b, a * b,
len(a) + 3, [5, 9], [5, 9].append(3), [5, 9, 3][2]*4,
[5, 9, 3].pop()*4, ([5, 9] + [3])[2]*4, ’hej’[0],
’hej’.count(’e’), a == 3, a == 3 and b > 4, ’hej’ in a,
’hej’ not in a, not ’hej’ in a, random.randint(5, 9),
t.forward(5), t.xcor(), t.xcor, f’{t.cxor()}’
```

- Vissa av exemplen är mer konstlade än andra
  - Du bör förstå alla!

UPPSALA UNIVERSITET

Värden och referenser

- `int`, `float`, `bool`, `str` beter sig som oföränderliga *primitiva värden*
- `list` och många andra typer är objekt av klasser
  - De flesta tillåter förändringar genom fält och/eller metoder
  - En variabel innehåller inte ett objekt, den refererar till ett objekt
  - En lista innehåller inte ett objekt, den refererar till ett objekt
    - Flera element i en lista kan referera till samma objekt
  - Jämför OU2 – om du gör `append` på en lista påverkas *alla* referenser till den listan
- En `tuple` i sig är också oföränderlig, men objekt en `tuple` refererar till kan förändras
- Speciella värdet `None`, en referens till ingenting

UPPSALA UNIVERSITET

Satser

- Villkor

```
if uttryck:
        kodblock
elif uttryck:
        kodblock
else:
        kodblock
```

- Inget krav att ha `elif` eller `else`!

UPPSALA UNIVERSITET

Satser

```
while uttryck:
        kodblock
```

Som if, men flera gånger. (else efter while körs sista gången när villkoret blir falskt, om man inte gjort break, lite klurigt)

UPPSALA UNIVERSITET

for

- Allt kan uttryckas med `while`. `for` är snyggare, förklarar mer vad man vill göra.

```
for tilldelningsmål in itererbart objekt:
    kodblock
```

- Iterera över ett objekt, tilldela varje element till tilldelningsmålet
- Enklast

for *variabel* in *lista*:

Men också t.ex:

```
for a, b in lex.items():
```

UPPSALA UNIVERSITET

Viktiga itererbara objekt

- Funktionerna `range`, `enumerate`, `zip`

`range(5)` `0, 1, 2, 3, 4` (inte 5)

`range(2, 5)` `2, 3, 4`

`range(2, 12, 3)` `2, 5, 8, 11`

```
enumerate(['a', 'b', 'c'], start = 2)
      (2, 'a'), (3, 'b'), (4, 'c')
zip([1, 2, 3], [4, 5, 6], [7, 8, 9])
      (1, 4, 7), (2, 5, 8), (3, 6, 9)
```

UPPSALA UNIVERSITET

Viktiga itererbara objekt

- Iterera över öppna filer ger deras rader
- Iterera över `re.finditer` ger träffarna på ett reguljärt uttryck i en sträng, en i taget
- Iterera över ett lexikon `lex` ger **bara nycklarna**
- Iterera över `lex.keys()` ger också bara nycklarna
- Iterera över `lex.values()` ger bara värdena
- Iterera över `lex.items()` ger tupler (*`nyckel, värde`*)
- I Python 3.7 och senare ger `keys`, `values`, `items` alltid värden i den ordning nycklarna först stoppades in i lexikonet
  - Så man *kan* ta bort den första raden i OU4 med `del lex[lex.keys()[0]]`
  - Gäller inte i Pythonversioner före 3.6, gäller inte andra ”lexikonliknande” objekt

UPPSALA UNIVERSITET

`break`, `pass`, `continue`

- `break` avslutar en loop (i förtid)
  - Återstående värden passeras inte, `while`-villkoret kontrolleras inte igen för en `while`-loop
- `continue` hoppar till *nästa* steg i en loop
  - Resten av kodet i blocket körs inte för den nuvarande *iterationen*
  - Hämtar nästa värde och början om från början för `for`, testar villkoret igen för `while`
- `pass` gör **ingenting**, används i tomma block

UPPSALA UNIVERSITET

Indexering

- Index för *sekvenser* (listor, strängar, tupler) börjar med 0
- Längden kan fås med `len(`*`sekv`*`)`
- Negativa index kan användas för att börja bakifrån

  `sekv[-1]` betyder sista elementet

  `sekv[-len(sekv)]` ett krångligt sätt att säga första elementet
- `sekv[4]` för att hämta element 4
- `sekv[4] = x` för att sätta element 4 till x
- `del sekv[4]` för att ta bort element 4

UPPSALA UNIVERSITET

Skivning

- Ur en sekvens kan man *skiva (slice)* ett delintervall

`sekv[start:end:step]`

`sekv[:]` kopia på hela sekvensen

`sekv[2:]` från element två och framåt

`sekv[:2]` fram till elementet före två (ett)

`sekv[2:2]` således tom lista

`sekv[1:9:3]` sekvens med element 1, 4, 7

`sekv[-8:]` de åtta sista elementen i sekv

`sekv[2:-8]` från element två till och med det nionde sista elementet (om det inte kommer före element två)

UPPSALA UNIVERSITET

Skivning

- Skivning ”slår i taket” och ”bottnar”
- Om sekv har mindre än 50 element kommer `sekv[:50]` att ge hela listan, utan något fel. Bara `sekv[50]` ger ett fel
- `sekv[-50:]` också, `sekv[-50:2]` blir då samma sak som `sekv[:2]`
  - Kunde användas i kompakt `smooth_b`

UPPSALA UNIVERSITET

Indexering av lexikon

- Vissa andra typer kan också indexeras
- Lexikon indexeras på sin nyckel
  - Ofta en sträng
  - Kan vara tal eller många andra typer också

```
lex['hej'] = 'ett bra värde för hej'
print(lex['hej'])
```

- Vi kan inte skiva lexikon.

UPPSALA UNIVERSITET

Definiera lexikon och listor direkt

- Vi kan beskriva en lista genom att skriva dess element
  `[1, 9, 'hej', None]`
- Vi kan beskriva ett lexikon genom paren av nycklar och värden
  `{'hej' : 5, 'nej' : False}`
- Vi kan definiera en lista genom att beskriva en iteration som skapar listan, en listbyggare (*list comprehension*)

  `[i + 5 for i in range(-5, 5) if i > 0]` ger listan

  `[6, 7, 8, 9]`
- Motsvarande för lexikon.

UPPSALA UNIVERSITET

Definiera en tupel

- En tupel definieras med en enkel uppräkning `5, 6, 7`
- När den kan misstas för något annat sätter man parentes runtomkring `(5, 6, 7)`
- `(5,)` är en tupel med ett enda värde
- `(,)` är en tupel med noll värden

UPPSALA UNIVERSITET

Andra sätt

- Initierarmetoderna för list och dict kan ta ett itererbart objekt som parameter och skapa en lista respektive ett lexikon utifrån det
  - När vi gör `list(lex.items())` är det precis det vi gör
    - `lex.items()` returnerar ett itererbart objekt
    - Initierarmetoden för `list` itererar över det objektet och skapar de elementen i lista

UPPSALA UNIVERSITET

Strukturera vår kod

- Funktioner
- Klasser

- Kapsla in logik för att göra varje nivå av koden läslig och hanterlig
- `vietnamese_flag` använde `pentagram` och `rectangle`, som använder klassen `Turtle`

UPPSALA UNIVERSITET

Skapa en funktion

```
def funktion(parametrar):
        kodblock
```

- En funktion kan returnera ett värde med satsen `return`
- Funktionen slutar köras när man returnerar.
- Värdet kan vara en tupel, som ju i praktiken är flera värden.

```
return 5, 9
return 'Hej'
```

Om inget `return` innebär det att returvärdet är `None`.

UPPSALA UNIVERSITET

Lokala variabler

- Parametrar och andra variabler vi skapar i en funktion är *lokala*
- De påverkar inte vilka variabler som finns utanför funktionen.
- Om de *refererar till samma objekt* som finns i en variabel utanför påverkas förstås det objektet.
- Man kan *komma åt* variabler som finns utanför funktionen.
  - Men inte tilldela dem.
  - Använd normalt parametrar för det du vill komma åt.

UPPSALA UNIVERSITET

Standardvärden

```
def x(val = 3, val2 = 3):
      print(val, val2)
```

`x()` skriver ut 3 3

`x(5)` skriver ut 5 3

`x(val2 = 5)` skriver ut 3 5

- Värdena vi ger parametrarna när funktionen anropas kallas dess argument.

UPPSALA UNIVERSITET

Funktioner är också uttryck

- Funktioner kan lagras i variabler

`x = len`

`x([1, 4, 8])` returnerar 3

- Vi har sett hur funktionerna `sort` (gör om en lista så den blir sorterad) och `sorted` (skapa en sorterad lista från ett itererbart objekt) tar en parameter key
  - `key` ska vara en funktion som tar emot element från följden som sorteras och returnerar ett nytt värde som är det som faktiskt används i jämförelsen

UPPSALA UNIVERSITET

Docstring

- I början av en funktion, klass, modul sätter vi en sträng, av konvention flerradig (trippelcitationstecken) som kort förklarar vad den gör, saker att tänka på
  - I vilka fall kan en funktion användas, vad betyder dess parametrar
  - Förklara inte exakt hur koden gör något, förklara vad den är till för

```
def hej(n):
        """Skriver hej på skärmen n gånger,
           n ska vara ett icke-negativt heltal."""
        for _ in range(n):
              print('hej')
```

UPPSALA UNIVERSITET

Klasser

- Vi kan skapa egna klasser

```
class MinKlass:
      pass
```

- Skapa ett `MinKlass`-objekt precis som du kan skapa en tom lista

```
x = MinKlass()
```

UPPSALA UNIVERSITET

Metoder

- Klasser kan ha olika metoder

```
class MinKlass:
      def hej(self):
            print(’hej’)
            self.val = 1

x = MinKlass()
x.hej()
print(x.val)
```

UPPSALA UNIVERSITET

Magiska metoder

- Magiska metoder har speciella roller
  - Används för olika funktionalitet i Python, som jämförelse, iterera, indexering
- Vi har främst tittat på initeringsmetoden `__init__` och strängmetoden `__str__`
- Initieringsmetoden anropas när instansen skapas

```
class MinKlass:
    def __init__(self, instr = ’hej’):
        if instr == ’hej’:
            self.val = 1
        else:
            self.val = 0
```

UPPSALA UNIVERSITET

Flera instanser av klassen, flera objekt

```
x = MinKlass()
y = MinKlass('hej')
z = MinKlass('bye')
print(x.val, y.val, z.val)
z.hej()
print(x.val, y.val, z.val)
```

UPPSALA UNIVERSITET

__str__

- Anropas när vi vill göra en sträng av vårt objekt
- Som i `print` och formatsträngar

```
def __str__(self):
    return f’VAL: {self.val}’

print(x, y, z)
```

UPPSALA UNIVERSITET

Nästa steg?

- Vad vill du automatisera?
  - Gör hela labbrapporter i Jupyter, spara som HTML och skriv ut (som sidan för lektion 9)
  - Lätt att generera nya grafer om värden ändras, många identiska grafer för olika dataset, filtrera bort outliers automatiskt…
- Skriva en app i Python?
  - Tyvärr enklare för Android än iOS (svårare att få installera egengjorda appar för Apple)
  - Kolla in paketen Kivy och Beeware

UPPSALA UNIVERSITET

Lycka till!