

Föreläsning 8

Klasser och objektorientering

Chalmers (DAT455) och GU (DIT001)

Jonas Duregård

Dagens ämne

- Att skriva egna klasser med konstruktor, attribut och metoder
- **self** och den inre vyn av klasser
- Inkapsling och invarianter
- Exempelklasser: Person och Racer

Kom ihåg: Använd en klass utifrån

- Vi har sett ett par klasser, till exempel Point som representerar koordinater i ett 2D-fönster

- Exempelkod:

`p = Point(2, 4)`

Använd konstruktorn Point för att skapa ett objekt

`p.move(1, 0)`

Flytta p ett steg i x-led

`print(p.getX())`

Skriv ut x-attributet för p (här 3)

Objektifiera personer

- Ett klassiskt nybörjarexempel: En klass där varje objekt representerar en person
- En bra start för att skriva en klass är att bestämma vad man vill kunna skriva för kod när klassen är klar
- Jag vill kunna skriva så här:

```
p = Person("Jonas", "Duregård")
print(p.getFirstName())
print(p.getFullName())
```
- Och få först utskriften "Jonas", sedan "Jonas Duregård"
- Klassen behöver alltså:
 - En konstruktor som tar för- och efternamn
 - Metoder getFirstName() och getFullName() för att ge förnamn respektive för- och efternamn

Klassdefinition och konstruerare

- Så här ser Python-koden ut för att skapa en klass Person med en konstruerare som tar två parametrar och sparar dessa i klassvariabler

```
class Person:  
    def __init__(self, firstName, lastName):  
        self.fName = firstName  
        self.lName = lastName
```

En funktionsdefinition
inuti klassen!

- Mycket magi på en gång här!
 - Nyckelordet class (som skapar en klass)
 - Funktionsdefinitionen __init__ som definierar en konstruerare
 - Variabeln self som är ett alias för objektet som håller på att skapas

Skapa Person-objekt

- Här skapar vi två Person-objekt och ser att de får olika attributvärden beroende på konstruktorns parametrar
- I första anropet till Person() är self ett alias för p1, i andra för p2

```
class Person:  
    def __init__(self, firstName, lastName):  
        self.fName = firstName  
        self.lName = lastName
```

```
p1 = Person("Jonas", "Duregård")  
p2 = Person("Amanda", "Duregård")
```

```
print(p1.fName)  
print(p2.fName)
```

Första anropet sätter den här raden
p1.fName, andra anropet p2.fName

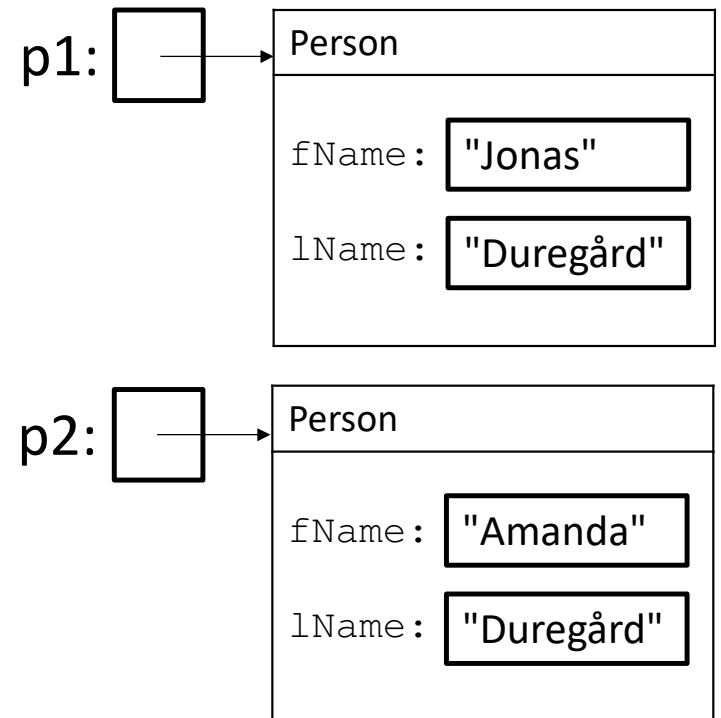
Skriver ut "Jonas" och "Amanda" (men
inte på det sättet vi vill kunna göra det!)

Person-objekten i minnet

- Med samma notation som i boken (pilar visar referenser)
- Det här visar sluttillståndet efter att båda objekten skapats, därför finns inte variabeln `self` från konstruktorn längre

```
class Person:
    def __init__(self, firstName, lastName):
        self.fName = firstName
        self.lName = lastName

p1 = Person("Jonas", "Duregård")
p2 = Person("Amanda", "Duregård")
```



Lägg till getter-metoderna

- Som vi bestämt skulle vi ha två metoder `getFirstName` och `getFullName`
- Metoder i klasser ser precis ut som vanliga funktionsdefinitioner, med en extra parameter `'self'` som är objektet metoden körs för

```
class Person:
```

```
    def __init__(self, firstName, lastName):  
        self.fName = firstName  
        self.lName = lastName
```

```
    def getFirstName(self):  
        return self.fName
```

```
    def getFullName(self):  
        return self.fName + " " + self.lName
```

Båda metoderna har egentligen
noll parametrar

Self används för att komma åt
klassvariabler/attribut

Klassen hittills + användning

```
class Person:
    def __init__(self, firstName, lastName):
        self.fName = firstName
        self.lName = lastName

    def getFirstName(self):
        return self.fName

    def getFullName(self):
        return self.fName+" "+self.lName
```

```
p = Person("Jonas", "Duregård")
print(p.getFirstName())
print(p.getFullName())
```

Utskrift:

Jonas

Jonas Duregård

Kör getFirstName med self=p
(self blir ett alias för p)

Self blir det som står före punkten!

```
class Person:
    def __init__(self, firstName, lastName):
        self.fName = firstName
    ...

    def getFirstName(self):
        return self.fName
```

```
p1 = Person("Jonas", "Duregård")
p2 = Person("Amanda", "Duregård")
print(p1.getFirstName())
print(p2.getFirstName())
```

Här blir self=p1 i getFirstName,
så p1.fName skrivs ut (Jonas)

Här blir self=p2 i getFirstName,
så p2.fName skrivs ut (Amanda)

Använd self för att anropa metoder

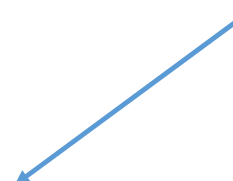
- Förutom att self används i metoder för att hämta/ändra värden, så kan man anropa metoder från klassen

```
class Person:
    def __init__(self, firstName, lastName):
        self.fName = firstName
        self.lName = lastName

    def getFirstName(self):
        return self.fName

    def getFullName(self):
        return self.getFirstName()+" "+self.lName
```

Här har jag ändrat från att använda self.fName till att använda metoden getFirstName(), med samma resultat



Bra design: Undvik redundans

- Man skulle kunna tänka sig att ha både firstName och fullName som variabler i klassen, så getFullName bara returnerar ett variabelvärde
- Detta vore dålig design: Om firstName ändras måste vi också ändra fullName
 - Uppstår lätt buggar om vi ändrar det ena men glömmer ändra det andra
- Om ett attribut går att beräkna från andra attribut är det oftast bäst att inte ha en egen variabel för det utan räkna ut det vid behov
- Exempel: Om person hade getAge() och getBirthYear() skulle det också uppstå redundans och möjligen inkonsekventa värden.
 - Ålder beräknas enklast utifrån födelsedatum och nuvarande datum

Funktioner vs. metoder

Två sätt att tillåta namnändringar av personer:

- Med en funktion:

```
def setFirstName(person, name):  
    person.fName = name
```

- Med en metod:

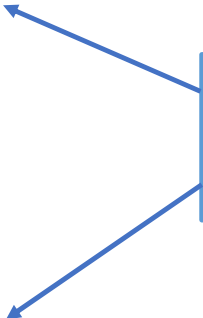
```
class Person:  
    ...  
    def setFirstName(self, name):  
        self.fName = name
```

- Användning (funktion/metod):

```
setFirstName(px, "Amanda")
```

```
px.setFirstName("Amanda")
```

Identiska förutom namnet
på första parametern!



self och den inre vyn av en klass

- När vi uteifrån anropar `p.doStuff()` säger vi åt `p` att göra något. Vi vill veta så lite som möjligt om hur `p` gör det, bara att det blir gjort
- När vi inuti klassen definierar `doStuff(self)` är det "jag" som blir tillsagd att göra något, och jag använder mig av klassvariabler och så vidare som jag behöver för att utföra uppgiften
- Exempel:
 - Utifrån sett är det enda jag vet att om jag kör `p.setFirstName("Jonas")` så kommer `p.getFirstName()` ge strängen "Jonas". Hur det går till bryr jag mig inte om.
 - Inuti klassen vet jag att `setFirstName` ändrar variabeln `self.fName` och att `getFirstName` returnerar den.

Exempel: Personer som kan gifta sig

- Vi vill lägga till en metod för att gifta en person med en annan person
- Det finns ett par regler runt giftermål:
 - Man får bara gifta sig om man inte redan är gift
 - 'Gift med' är en symmetrisk relation: Om a är gift med b så är b gift med a
 - Den är också antireflexiv: Man får inte vara gift med sig själv (Det är sant!)
- Vi vill designa vår klass så att dessa regler alltid efterföljs

Yttre vy

- Jag vill kunna köra det här programmet

spouse: engelska för maka/make

```
p1 = Person("Jonas", "Duregård")  
p2 = Person("Amanda", "Duregård")
```

```
p1.marry(p2)
```

Gift p1 med p2 (samma sak som p2.marry(p1))

```
def marriageStatus(p):  
    if (p.isMarried()):  
        spouse = p.getSpouse()  
        return p.getFullName() + " is married to " + spouse.getFullName()  
    else:  
        return p.getFullName() + " is not married"
```

Är p gift?

p.getSpouse() ska ge ett Person-objekt!

```
print(marriageStatus(p1))  
print(marriageStatus(p2))
```

- Utskrifterna ska bli "Jonas Duregård is married to Amanda Duregård" / "Amanda Duregård is married to Jonas Duregård"

Nya metoder

Tre nya metoder i Person behövs:

- marry används för att gifta två personer
- isMarried ger ett boolean-värde som svarar på om en person är gift
- getSpouse ger personen en person är gift med (som ett Person-objekt!)

```
p1 = Person("Jonas", "Duregård")  
p2 = Person("Amanda", "Duregård")
```

→ `p1.marry(p2)`

```
def marriageStatus(p):  
    if (p.isMarried()):  
        spouse = p.getSpouse()  
        return p.getFullName() + ...  
    else:  
        return p.getFullName() + ...
```

```
print(marriageStatus(p1))  
print(marriageStatus(p2))
```

Att använda None

- None är ett särskilt objektsvärde i Python
- Kan användas för variabler som ibland inte har något värde

```
class Person:
    def __init__(self, firstName, lastName):
        self.fName = firstName
        self.lName = lastName
        self.spouse = None
    ...
    def isMarried(self):
        return self.spouse is not None

    def getSpouse(self):
        return self.spouse
```

Alla Person-objekt har en spouse-variabel,
men den är None för icke-gifta

"x is not None" ger True om x inte är None

getSpouse är en enkel getter-metod

Ett första försök för marry

- När `p1.marry(p2)` körs har vi två person-objekt (`self` och `other`) i den inre vyn
- Vi måste sätta `p1` till att vara gift med `p2`, och `p2` med `p1`

```
class Person:
    def __init__(self, firstName, lastName):
        self.fName = firstName
        self.lName = lastName
        self.spouse = None

    ...

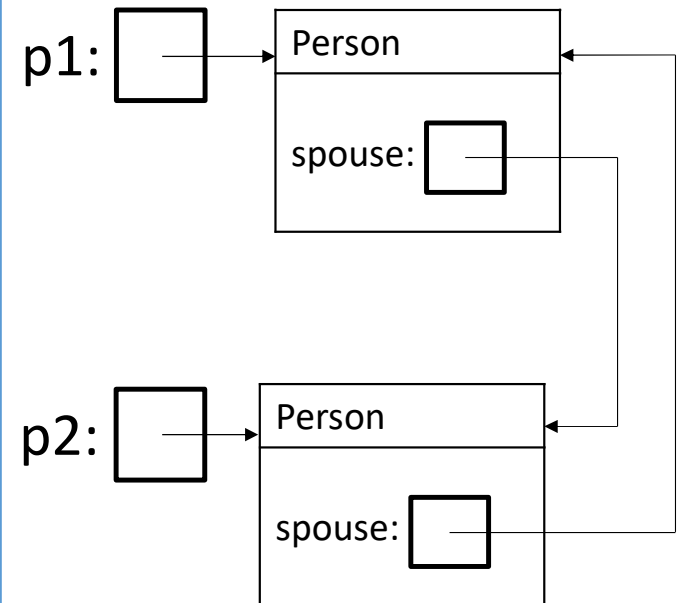
    def isMarried(self):
        return self.spouse is not None

    def getSpouse(self):
        return self.spouse

    def marry(self, other):
        self.spouse = other
        other.spouse = self
```

self är en variabel!

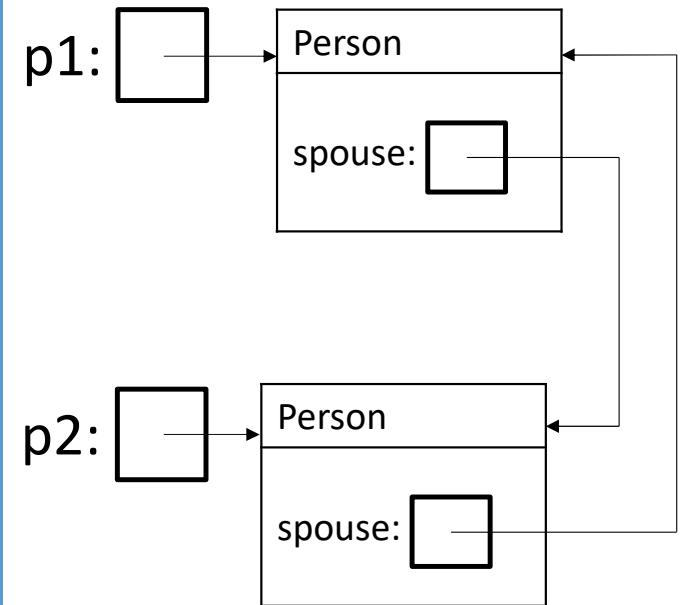
Efter `p1.marry(p2)`:



Två objekt som refererar till varandra

- Efter att vi kört `p1.marry(p2)` så är `p1.spouse==p2` och `p2.spouse==p1`
- `p1` innehåller alltså `p2` och `p2` innehåller `p1`!
- Det här fungerar eftersom `p1.spouse` och `p2.spouse` är referenser, objekten *innehåller* alltså egentligen inte varandra utan *refererar* till varandra
- Ett exempel på när aliasing är nödvändigt, ifall en variabel skulle vara dess värde skulle två objekt inte kunna innehålla varandra!

Efter `p1.marry(p2)`:



Kul grej

- Vad får jag om jag kör det här:

```
p1.marry(p2)
```

```
x = p1.getSpouse().getSpouse().getSpouse().getSpouse().getFirstName()
```

p1.getSpouse() ger p2

p2.getSpouse() ger p1

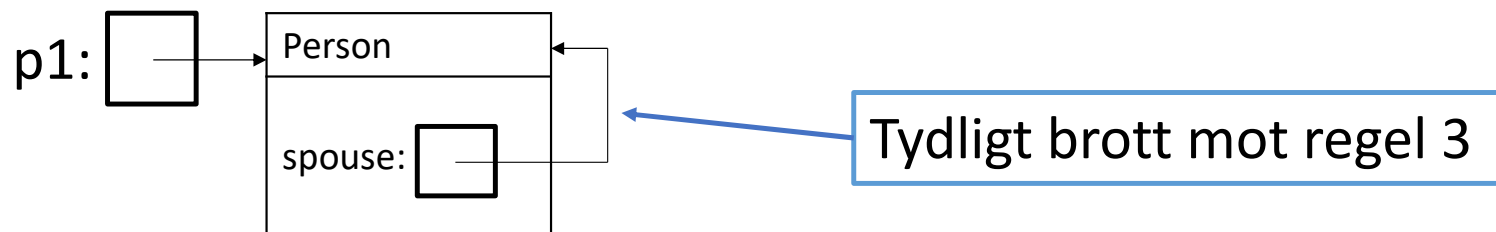
p1.getSpouse() ger p2

p2.getSpouse() ger p1

Hela uttrycket ger samma resultat som p1.getFirstName()!

Problem med marry

- Som marry är nu kan man bryta mot alla tre regler vi satte upp tidigare
- Exempelvis om vi kör `p1.marry(p2)` sedan `p2.marry(p3)` kommer både p1 och p3 vara gifta med p2, som dock bara är gift med p3
- Dessutom är `p1.marry(p1)` fullt möjligt, då kommer marry köras med två variabler som är alias för samma personobjekt



En bättre marry-metod

- Vi avbryter metoden om någon av parterna redan är gift eller om de är samma person
- Vi returnerar True/False för att visa för anroparen om giftermålet lyckades

```
def marry(self, other):  
    if (self.isMarried() or other.isMarried()):  
        return False  
    if (self == other):  
        return False  
  
    ← self.spouse = other  
    other.spouse = self  
    return True
```

Notera att det inte behövs någon else, eftersom metoden avbryts vid return

När körningen hit vet vi att self och other är två olika ogifta personer

Inkapsling (encapsulation)

- Genom att tänka efter hur vi designade klass ser vi till att oavsett vilka metoder som anropas kommer vissa regler (kallas 'invarianter') alltid upprätthållas
- Oavsett hur slarvig en användare är kommer den inte kunna använda metoderna för att bryta mot någon av reglerna
- En viktig del av det är att vi inte har någon `setSpouse()`-metod eller liknande
 - Vi gömmer avsiktligt undan funktionalitet som skulle kunna användas för att bryta reglerna och exponerar enbart metoden `marry()` för att ändra vem som är gift med vem
 - Vi säger att vi kapslat in spouse-attributet

Invarianter i Person

- Vi kan uttrycka reglerna Person följer med Python-kod
- Reglerna använder helt det yttre gränssnittet (metoderna som en användare av klassen har tillgång till)
- För alla personer `p` gäller: `p.getSpouse() != p`
(ingen person är gift med sig själv)
- För alla gifta personer `p` gäller:
 - `p.isMarried()`
 - `p.getSpouse().getSpouse() == p`
(personen `p` är gift med är också gift med `p`)
- För ogifta personer gäller: `p.getSpouse() is None`

Person i helhet

- Teknisk analys:
 - Konstruktor som tar två strängar
 - 3 attribut/klassvariabler
 - 6 metoder
 - 4 tar inga parametrar
 - 2 tar en parameter var
 - 1 ger inget resultat
 - 2 ger boolean-resultat
 - 2 ger sträng-resultat
 - 1 ger ett Person-objekt
 - Alla utom marry är enkla "one-liners"

```
class Person:
    def __init__(self, firstName, lastName):
        self.fName = firstName
        self.lName = lastName
        self.spouse = None

    def getFirstName(self):
        return self.fName

    def setFirstName(self, name):
        self.fName = name

    def getFullName(self):
        return self.fName + " " + self.lName

    def isMarried(self):
        return self.spouse is not None

    def getSpouse(self):
        return self.spouse

    def marry(self, other):
        if (self.isMarried() or other.isMarried()):
            return False
        if (self == other):
            return False

        self.spouse = other
        other.spouse = self
        return True
```

Fundera på

- Var i klassen Person ...
 - ... undviker vi redundans?
 - ... använder vi inkapsling?
 - ... finns exempel på aliasing/referenser?

Kom ihåg: Racer.py

- Förra föreläsningen använde vi klasser från graphics.py för att flytta runt en liten bil (en blå cirkel)
- racer.py finns på kurshemsidan
- Att göra bilen snyggare skulle bli ganska rörigt
 - Just nu *är* bilen ett cirkel-objekt, skulle den bestå av flera grafikobjekt måste alla objekt flyttas samtidigt
- Kanske kan vi göra det enklare genom att skapa en klass för racerbilen?

```
from graphics import *

speedLimit = 5
animationSpeed = 50

win = GraphWin("Racer" , 640, 480, autoflush=False)
win.setCoords(-320, -240, 320, 240)

racer = Circle(Point(0,0), 10)
racer.setFill('blue')
racer.draw(win)

xvelocity = 0
yvelocity = 0

while(True):
    key = win.checkKey() # The key being pressed (if any)

    if key == "Up":
        yvelocity += 1
    elif key == "Down":
        yvelocity -= 1
    elif key == "Right":
        xvelocity += 1
    elif key == "Left":
        xvelocity -= 1
    elif key == "space":
        # Break by 10% each animation step if holding space
        xvelocity *= 0.9
        yvelocity *= 0.9
    elif key == "BackSpace":
        # Move the racer back to (0,0) and stop it
        racer.move(-racer.getCenter().getX(),-racer.getCenter().getY())
        xvelocity = 0
        yvelocity = 0

    # Enforce the speedlimit
    yvelocity = min(speedLimit, yvelocity)
    yvelocity = max(-speedLimit, yvelocity)
    xvelocity = min(speedLimit, xvelocity)
    xvelocity = max(-speedLimit, xvelocity)

    # Move the racer based on current velocity
    racer.move(xvelocity,yvelocity)

    update(animationSpeed)
```

racer2.py: Prototyp

- Precis som med Person gör vi en prototyp på ett program som använder klassen vi vill skapa
 - Prototypen är baserad på huvudloopen från racer.py
 - Vi flyttar in alla attribut hos bilen (hastighet, position, grafik ...) i klassen
 - Vi byter ut all kod som manipulerar bilen på något sätt mot anrop

racer2.py: Prototyp för huvudloop

```
while(True):  
    key = win.checkKey()  
    if key == "Up":  
        racer.accelerate(0,1)  
    elif key == "Down":  
        racer.accelerate(0,-1)  
    elif key == "Right":  
        racer.accelerate(1,0)  
    elif key == "Left":  
        racer.accelerate(-1,0)  
    elif key == "space":  
        racer.slowDown()  
    elif key == "BackSpace":  
        racer.reset()  
  
    # Move based on current veclocity  
    racer.drive()  
  
    # Wait for animation to finish  
    update(animationSpeed)
```

- racer är ett Racer-objekt
- Metoder vi behöver i klassen Racer:
 - accelerate för att ändra hastighet
 - slowDown för att bromsa mjukt
 - reset för att starta om
 - drive för att köra "en tidsenhet"
- Det här är så klart bara en av många möjliga prototyper
- Ofta behöver prototyper justeras för att något visar svårt att implementera

racer2.py: Skelett för klassen

- Baserat på vår prototyp vet vi att klassen kommer se ut ungefär så här:

```
class Racer:
```

```
    def __init__(self, ...):
```

```
        ...
```

```
    def accelerate(self, xacc, yacc):
```

```
        ...
```

```
    def slowDown(self):
```

```
        ...
```

```
    def drive(self):
```

```
        ...
```

```
    def reset(self):
```

```
        ...
```

Vi har inte bestämt parametrar för konstruktorn

Accelerate tar två parametrar, alla andra metoder tar noll parametrar (self räknas inte)

racer2.py: Klass med konstruerare

- Vi skapar klassen och flyttar in koden för att initialisera grafiken och hastighetsvärden
- Här har jag valt att inte skapa fönstret i konstruktorn, utan istället ta det som parameter

Vi tar fönstret som bilen ska ritas i som parameter

```
class Racer
    def __init__(self, win):
        self.circle = Circle(Point(0,0), 10)
        self.circle.setFill('blue')
        self.circle.draw(win)

        self.xvelocity = 0
        self.yvelocity = 0
```

Klassen har tre attribut:
circle, xvelocity, yvelocity

racer2.py: Initialisering

- Utanför klassen, innan huvudloopen ersätter vi koden som skapade cirkeln med ett anrop till konstruktorn för klassen Racer

```
win = GraphWin("Racer" , 640, 480, autoflush=False)
# This line sets (0,0) to be the middle of the window
win.setCoords(-320, -240, 320, 240)
```

```
racer = Racer(win)
```

```
# Main animation loop
while(True):
    key = win.checkKey()
    if key == "Up":
```

Från klassdefinitionen:

```
class Racer
    def __init__(self, win):
```

racer2.py: accelerate och slowDown

- Metoderna är väldigt lika koden vi använde innan, fast vi har lagt till 'self.' på över 9000 ställen (bildligt talat)
- Det är väldigt lätt att missa användningen av self!

```
class Racer:
```

```
...
```

```
def accelerate(self, xacc, yacc):
```

```
    self.xvelocity += xacc
```

```
    self.yvelocity += yacc
```

```
    # Enforce the speedlimit
```

```
    self.yvelocity = min(speedLimit, self.yvelocity)
```

```
    self.yvelocity = max(-speedLimit, self.yvelocity)
```

```
    self.xvelocity = min(speedLimit, self.xvelocity)
```

```
    self.xvelocity = max(-speedLimit, self.xvelocity)
```

```
def slowDown(self):
```

```
    self.xvelocity *= 0.9
```

```
    self.yvelocity *= 0.9
```

speedLimit är en konstant utanför klassen



racer2.py: Bekvämlighetsmetoder

- Det är fritt fram för oss att lägga till metoder utöver de vi behöver i huvudloopen
- Två ganska praktiska metoder vore att hämta x- och y- koordinater för bilen
- Det här är ett exempel på "getters" som inte direkt returnerar en klassvariabel, men som logiskt sett ger ett attribut för bilen

```
class Racer:
    ...
    def getX(self):
        return self.circle.getCenter().getX()

    def getY(self):
        return self.circle.getCenter().getY()
```

racer2: drive och reset

- Både drive och reset ska flytta bilen
 - drive flyttar den baserat på nuvarande hastighet och position
 - reset flyttar tillbaks den till (0,0)
- Jag vill undvika att skriva två olika koder som flyttar den grafiska komponenten

```
...  
elif key == "BackSpace":  
    racer.reset()
```

Om man trycker backspace ska bilen återgå till (0,0) och stanna

```
# Move based on current veclocity  
racer.drive()
```

Varje steg i animationen ska bilen flytta sig baserat på nuvarande hastighet

```
# Wait for animation to finish  
update(animationSpeed)
```

racer2: Interna metoder

- En möjlighet är att skapa två metoder: En för att flytta bilen till en absolut (x,y)-position och en för att flytta relativt till dess nuvarande position
- Genom att räkna om absoluta positioner till relativa sker all förflyttning på ett enda ställe

```
class Racer:
```

```
...  
def _moveAbsolute(self, x,y):  
    self._moveRelative(x-self.getX(),y-self.getY())
```

```
def _moveRelative(self, xDiff,yDiff):  
    self.circle.move(xDiff, yDiff)
```

'_' innan ett metodnamn
antyder att metoden endast
är för internt bruk i klassen

_moveAbsolute anropar
_moveRelative

Den enda koden som
faktiskt flyttar något

racer2: Åter till drive och reset

- Med `_moveAbsolute` och `_moveRelative` kan vi ge enkla och tydliga definitioner för drive och reset

```
class Racer:
```

```
...
```

```
def drive(self):
```

```
    self._moveRelative(self.xvelocity, self.yvelocity)
```

```
def reset(self):
```

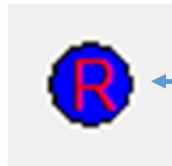
```
    self._moveAbsolute(0, 0)
```

```
    self.xvelocity = 0
```

```
    self.yvelocity = 0
```

racer2: Tillbaks där vi började?

- Efter alla dessa ändringar fungerar bilen exakt som den gjorde i racer.py
- Skillnaden är att vi har en bättre design, som är lättare att utöka!
- Så låt oss nu ändra hur bilen ser ut, jag vill förbättra grafiken för racerbilen med ett häftigt rött R ovanpå den blå cirkeln



En riktigt snygg racerbil

racer2: Ändra grafiken

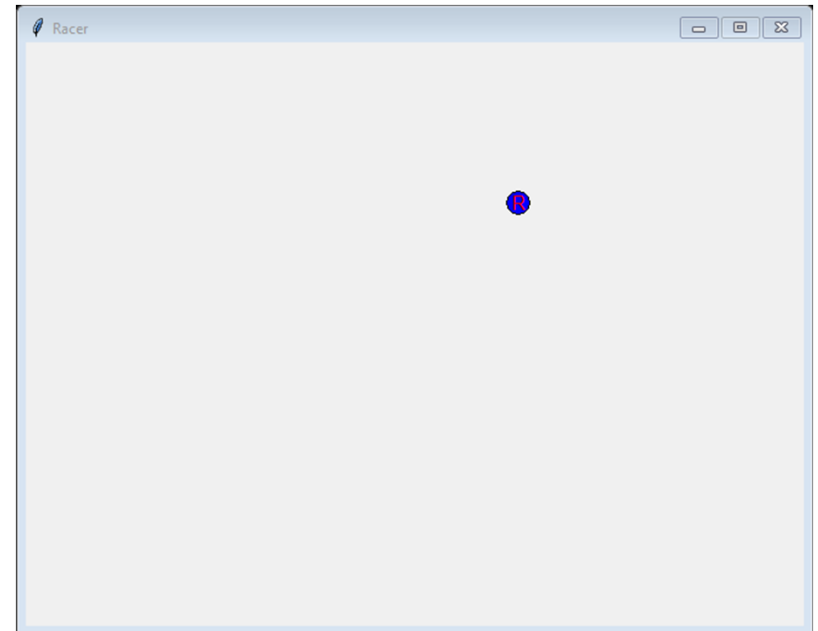
- För att lägga till en bokstav på bilens grafik behöver vi bara ändra två saker
 - Skapa och rita ut textelementet i konstruktorn
 - Ändra `_moveRelative` så bokstaven flyttas med cirkeln

```
class Racer:
    def __init__(self, win):
        self.circle = Circle(Point(0,0), 10)
        self.circle.setFill('blue')
        self.circle.draw(win)

        self.letter = Text(Point(0,0), "R")
        self.letter.setTextColor('red')
        self.letter.draw(win)

        self.xvelocity = 0
        self.yvelocity = 0

    def _moveRelative(self, xDiff, yDiff):
        self.circle.move(xDiff, yDiff)
        self.letter.move(xDiff, yDiff)
```



racer2: Fördelning av ansvar

- Den främsta vinsten med racer2.py jämfört med racer.py är hur "ren" koden för animeringsloopen blivit
- Klassen ansvarar helt för hur bilen ser ut
 - Vi säger ingenting om det i loopen!
- accelerate ansvarar för hastighetsgräns
- Klassen ansvarar för att hastighet och position räknas ut korrekt

```
while(True):
    key = win.checkKey()
    if key == "Up":
        racer.accelerate(0,1)
    elif key == "Down":
        racer.accelerate(0,-1)
    elif key == "Right":
        racer.accelerate(1,0)
    elif key == "Left":
        racer.accelerate(-1,0)
    elif key == "space":
        racer.slowDown()
    elif key == "BackSpace":
        racer.reset()

    # Move based on current veclocity
    racer.drive()

    # Wait for animation to finish
    update(animationSpeed)
```

Gör det rätt från början

- Här har vi visat hur vi först skapade ett program utan klasser, och sedan ändrade det till att använda en klass
- Ett bättre sätt är så klart att från början välja en bra design som är lätt att utöka med ny funktionalitet
 - Kräver mer planering/designarbete
 - Det är en svår avvägning mellan att å ena sidan komma igång snabbt med någonting och att å andra sidan att slippa göra om för mycket när designen visar sig ha problem

Är det värt besväret?

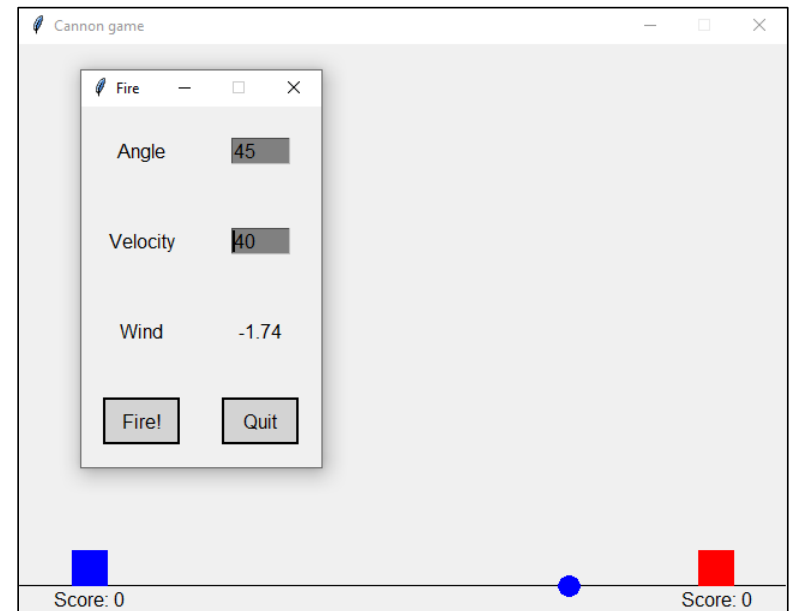
- En fråga som många nya programmerare ställer: Är objektorientering verkligen värt besväret?
- För enkla program som får plats på slides i en föreläsning är det svårt att visa hur viktigt det är med en bra design för att hålla ordning på kod
- Tar man sig an ett lite större programmeringsprojekt märker man snabbt hur viktigt det är att till exempel undvika att upprepa kod
 - Att tänka efter en stund för att göra koden lite enklare är oftast värt tiden det tar
 - Att ändra fungerande kod till en bättre struktur tar ofta emot lite, eftersom det riskerar att introducera buggar, men det är en väldigt bra vana eftersom koden oftast kommer behöva ändras ändå när ny funktionalitet läggs till

En ännu bättre design?

- En viktig princip när man programmerar grafik är att separera kod för *modell* och kod för *grafik*
- racer2.py följer inte riktigt den principen
 - Bilens position (modell-kod) är helt sammanknuten med kod för att rita ut ett cirkelobjekt (grafik-kod)
- Tanke: Gör en klass enbart för bilens modell och en klass enbart för grafik
- Varje objekt i grafikklassen är knutet till ett objekt i modellklassen, och dess enda jobb är att se till att grafiken som visas reflekterar modellen
- Det här är ungefär det ni kommer göra i labb 2!

Kort sammanfattning av labb 2

- Uppgiften i labb 2 är att skapa ett spel där två spelare ställer in riktning och hastighet på var sin kanon för att försöka träffa den andres kanon
- Kanonkulator påverkas av gravitation och vind
- Spelare får poäng för varje träff
- Designen ni ska använda för er kod är till största delen redan bestämd, men jag tänkte tala lite om hur man kan komma fram till en design på egen hand



racer3.py: Modellen för en racerbil

- Klassen Racer modellerar bilen som ett helt abstrakt objekt
- Hanterar position och hastighet men ingen grafik alls
- Vi kan "köra" runt bilen genom att anropa metoder och skriva ut värden för att testa att den fungerar

```
class Racer:
    def __init__(self):
        self.xvelocity = 0
        self.yvelocity = 0
        self.xpos = 0
        self.ypos = 0

    def _moveRelative(self, xDiff, yDiff):
        self.xpos += xDiff
        self.ypos += yDiff

    def _moveAbsolute(self, x, y):
        self._moveRelative(x-self.getX(), y-self.getY())

    def accelerate(self, xacc, yacc):
        self.xvelocity += xacc
        self.yvelocity += yacc
        # Enforce the speedlimit
        self.yvelocity = min(speedLimit, self.yvelocity)
        self.yvelocity = max(-speedLimit, self.yvelocity)
        self.xvelocity = min(speedLimit, self.xvelocity)
        self.xvelocity = max(-speedLimit, self.xvelocity)

    ...
```

racer3.py: Grafik för racerbilen

```
class RacerGraphics:
```

```
    def __init__(self, racer, win):
```

```
        self.win = win
```

```
        self.model = racer
```

```
        self.circle = Circle(Point(0,0), 10)
```

```
        self.circle.setFill('blue')
```

```
        self.circle.draw(win)
```

```
        self.letter = Text(Point(0,0), "R")
```

```
        self.letter.setTextColor('red')
```

```
        self.letter.draw(win)
```

```
    """
```

```
    Synchronize the graphics with the current state of the model
```

```
    """
```

```
    def sync(self):
```

```
        m = self.model
```

```
        c = self.circle
```

```
        t = self.letter
```

```
        # Move the circle and text elements to the current position of the model
```

```
        c.move(m.getX()-c.getCenter().getX(), m.getY()-c.getCenter().getY())
```

```
        t.move(m.getX()-t.getAnchor().getX(), m.getY()-t.getAnchor().getY())
```

Separat klass för grafik

Tar modellen som en parameter

En enda metod, som
synkroniserar grafik med modell

racer3.py: Huvudloop

```
# Create the main window.  
win = GraphWin("Racer" , 640, 480, autoflush=False)  
# This line sets (0,0) to be the middle of the window  
win.setCoords(-320, -240, 320, 240)
```

```
# Create a model of a race-car, and graphics for it
```

```
racer = Racer()
```

```
graphics = RacerGraphics(racer, win)
```

Skapar modell, och grafik för den separat

```
# Main animation loop
```

```
while(True):
```

```
    key = win.checkKey() # The key being pressed (if any)
```

```
    if key == "Up":
```

```
        ...
```

```
        # Move the racer based on current veclocity
```

```
        racer.drive()
```

```
        # Sync graphics with model
```

```
        graphics.sync()
```

```
        # Waits for the animation step to finish based on animationspeed, then redraws the window
```

```
        update(animationSpeed)
```

Synkroniserar grafiken i varje steg

Fördelar/nackdelar med den nya designen

- Klassen `Racer` har bara hand om abstrakta egenskaper hos bilen
 - Hur accelerar och saktar bilen ner, vilken position är den på osv.
 - Helt oberoende av grafikbiblioteket (`graphics.py`)
- Klassen `RacerGraphics` har hand om allt som beskriver hur bilen ser ut
 - Skapar grafikobjekt och ser till att de är där de ska vara
 - Tydligt ansvar: Se till att det som visas på skärmen stämmer med det som anges av modellen
- Nackdel: Vi måste minnas att anropa `sync()` så grafik och modell alltid är i synk
- Nackdel: Lite mer kod totalt sett

Föreläsningen

- Kapitel 10

QA-sessionen

- Öva på att skapa enkla klasser med konstruerare och metoder, samt att använda dessa