Föreläsning 7

Objekt och grafik

Introduktion till programmering i Python
Chalmers (DAT455) och GU (DIT001)

Jonas Duregård

Dagens ämne

- Att använda objekt och klasser
  - Att definiera egna klasser väntar vi med till nästa föreläsning
- Ett par tekniska aspekter av objektorientering
  - Använda konstruerare, attribut och metoder
  - Referenser och aliasing
- Att använda ett enkelt grafikbibliotek bestående av flera klasser
  - Dels som en övning på att arbeta med objekt och klasser, dels övning på att hantera enkel 2D-grafik i programmering

Nödvändiga verktyg

- Vi kommer använda graphics.py, ett väldigt enkelt grafikbibliotek som utvecklats speciellt för kursboken
  - Biblioteket och dess dokumentation kan ni hitta här: https://mcsp.wartburg.edu/zelle/python/
  - För att kodexemplen i dessa slides ska fungera behöver ni ladda ner biblioteket till samma mapp som .py-filen ni skriver och importera biblioteket med kodraden ”from graphics import *” (importera allt från modulen graphics)

Aspekter av objektorienterad programmering

- Den tekniska aspekten: Hur skapar man och manipulerar objekt i python?
- Design-aspekten: Själva anledningen till att vi använder objekt är att göra vår kod mer lättläst och enklare att använda.
  - Ett naturligt sätt att fördela ansvaret för funktionalitet, och svara på frågor i stil med ”var ska koden som gör det här placeras?”
- Jag kommer prata lite om båda delarna

”Avmystifiera” objektorientering

- Tyvärr är objektorientering ofta ganska förvirrande för nybörjare
- En del beskrivningar är väldigt abstrakta och svårbegripliga
- Det hjälper att följande fundamentala sanning i bakhuvudet hela tiden:

Ett objekt är i första hand ett sätt att samla ihop flera variabler

- Det är den mest korrekta korta beskrivningen av vad ett objekt är
- Ovanpå det lägger vi ett par behändiga sätt att manipulera variablerna i objekt
- Vidare ovanpå det lägger vi olika designprinciper och det är där någonstans det börjar bli krångligt

Objekt som data-behållare

- En första intuition för ett objekt: Slår ihop flera variabler till en, ungefär som en lista fast varje variabel har ett namn istället för en position
- Exempel, den här funktionen tar två parametrar (två person-objekt) men använder fyra variabler: p1.age, p2.age, p1.name, p2.name

```
def compareAges(p1, p2):
    if (p1.age == p2.age):
        return p1.name + " is the same age as " + p2.name
    elif (p1.age > p2.age):
        return p1.name + " is older than " + p2.name
    else: # p1.age < p2.age
        return p2.name + " is older than " + p1.name
```

Fortsättning om objekt som data-behållare

- I föregående exempel är p1 och p2 variabler (parametrar till en funktion) och p1.age, p2.age, p1.name och p2.name är variabler (objektsattribut)
- Vi kan använda attributen precis som andra variabler, inklusive tilldelningar

```
p1 = p2
```

Ändra hela p1 till ett annat objekt

```
p1.age = 35
```

Ändra age-attributet för p1 till 35

```
p2.name = "Jonas"
```

Ändra name-attributet för p2 till Jonas

Metoder – funktioner i objekt

- Objekt kan också innehålla funktioner
- Funktioner i objekt kallas *metoder*
- Det objektorienterade sättet att tänka är att metoder ”säger åt” objekt att göra saker
  - **`shout = text.upper()`**
    ”säg åt strängen text att skapa en kopia av sig själv med enbart versaler”
  - **`list.pop()`**
    ”säg åt listan list att ta bort sitt sista element”
- Notera skillnaden: upper() gör en ny sträng, pop() ändrar en existerande lista
- Både strängar och listor är objekt i Python, men de är lite ”magiska” för de är inbyggda i språket

Objekt som *representationer*

- Kärnan i objektorienterad design är att varje objekt *representerar* något från verkligheten
- Exempel: Säg att vi ska utveckla läroplattformen Canvas
  - Varje kurs representeras som ett objekt – med attribut som kurskod, en lista på studenter, labbuppgifter, sidor, …
  - Varje student representeras som ett objekt med namn, personnummer, e-post, …
  - Varje labbuppgifter representeras som objekt, med titel, brödtext, inlämningstyp, deadline ….
- Att tänka på vilka sorts objekt man behöver är ett viktigt steg i ett objektorienterat utvecklingsprojekt
- Med en bra design blir det tydligt var vi ska placera koden för varje funktion

Klasser

- En klass är som en mall för objekt
- Vi kan skapa klasser på ungefär samma sätt som vi skapar funktioner osv.
- Idag kommer vi enbart studera klasser ”utifrån”, alltså hur vi använder klasser som redan finns – inte hur vi skriver nya klasser
- Vi kommer också ta en första titt på ett par klasser från graphics.py:
  - **Point** som representerar en (x,y)-koordinat i ett 2D-fönster
  - **Circle** som representerar en cirkel med attribut som position, radie och färger för olika delar av cirkeln
- Klassnamn inleds alltid med stor bokstav

Konstruktorer (eng. Constructors)

- Varje klass har en konstruktor, som används för att skapa objekt
  - Exempel: Klassen Circle har konstruktorn Circle(centerpoint, radius) som skapar en cirkel med angiven mittpunkt och radie
  - Konstruktorn har alltid samma namn som klassen
  - Konstruktorns främsta uppgift är att initiera alla attribut för objektet

Skapa objekt med konstruerare

- Att använda en konstruerare ser precis ut som ett funktionsanrop, exempel

  ```
  Point(10,20)
  ```

- Anropet skapar ett objekt av klassen Point, 10 är och 20 är x- och y-koordinater
- Resultatet av ett anrop till konstruerare X(…) är alltid ett objekt av klassen X
- Två program som gör i princip samma sak:
  - ```
    p = Point(10,20)
    c = Circle(p, 5)
    ```
  - ```
    c = Circle(Point(10,20), 5)
    ```

Skapa en cirkel med mittpunkt (10,20) och radie 5

- I det andra programmet använder vi en konstruerare inuti en konstruerare!
- Anrop till konstruerare är uttryck, en bra tumregel är att du kan använda konstruerare överallt där du kan använda en variabel

Metoder i klasser

- Klasser innehåller metoddefinitioner
  - Vi kan exempelvis köra `c.move(3,4)` för ett Circle-objekt `c` på grund av att klassen Circle har en metod som heter move som tar två heltal (det metoden gör är att flytta cirkeln 3 steg i x-led och 4 i y-led

Notera: Just nu jobbar vi med cirklar och punkter som helt abstrakta representationer. När vi säger att cirkeln ”flyttas” menar vi inte att något rör sig på skärmen utan att attributen för Cirkel-objektet som beskriver dess position ändras!

Getters och setters

- I kursboken och många andra sammanhang undviker man att använda attribut ”utifrån” och använder metoder för att hämta och ändra attribut
  - v = x.getProp() sätter v till attributet prop från x, istället för v = x.prop
  - x.setProp(v) ändrar attributet prop till v för x, istället för x.prop = v
- Python hindrar oss inte från att göra på fel sätt. Exempel klassen Point:

```
p = Point(3,6)
print(p.getX()) # Prints 3
print(p.x)      # Also prints 3–but in a bad way
p.x = 5         # We probably shouldn't do this
```

- En anledning att ha getters och setters är att man vill att saker ska hända när attribut ändras
  - Exempel: **`c.setFill('red')`** ändrar färgen på en cirkel till röd, men har också effekten att måla om cirkeln ifall den är synlig i ett fönster

Offentliga gränssnitt

- En del av designen av en klass är att bestämma metoder som ska användas för att interagera med klassen
- Använder man något annat än det offentliga gränssnittet kanske klassen inte fungerar som det är tänkt
- Exempel, klassen Point (från ”Graphics Reference (graphics.py v5)”)

3.1 Point Methods

`Point(x,y)` Constructs a point having the given coordinates.
Example: `aPoint = Point(3.5, 8)`

`getX()` Returns the x coordinate of a point.
Example: `xValue = aPoint.getX()`

`getY()` Returns the y coordinate of a point.
Example: `yValue = aPoint.getY()`

- Det offentliga gränssnittet för Point har en konstruerare och två metoder för att hämta ut x-och y-värden (Point har en del ytterligare metoder som inte visas här)

Kom ihåg: Muterbarhet

- Strängar är icke-muterbara, om jag skriver x = "hello" så kommer x vara textsträngen "hello" fram tills jag ändrar den genom en tilldelning (x = …)
- Listor är muterbara, om jag skriver y = [1,2,3] kan jag ändra innehållet genom exempelvis y.push(4) – alltså utan någon tilldelning
- Både Point och Circle är muterbara, dels har de var sin move-metod som flyttar punkten/cirkeln och dels har de attribut som färger osv. som kan ändras

The module provides the following classes of drawable objects: `Point`, `Line`, `Circle`, `Oval`, `Rectangle`, `Polygon`, and `Text`. All objects are initially created unfilled with a black outline. All graphics objects support the following generic set of methods:

`setFill(color)` Sets the interior of the object to the given color.

Example: `someObject.setFill("red")`

`move(dx,dy)` Moves the object `dx` units in the x direction and `dy` units in the y direction. If the object is currently drawn, the image is adjusted to the new position.

Example: `someObject.move(10, 15.5)`

Sammanfattning:Enkel objektorientering

- Objekt har attribut/egenskaper/klassvariabler (olika namn för samma sak)
- Klasser är mallar för objekt som innehåller konstruerare och metoder
- Vi skapar objekt med konstruerare

```
p = Point(3,6)
```

Skapa ett point-objekt

- Vi interagerar med objekt genom metoder

```
p.move(1,0)
print(p.getX())
```

Flytta p ett steg i x-led

Skriv ut x-attributet för p

- Går att kombinera med allt annat ni lärt er hittils (tänk: en funktion med en loop som skapar en lista av objekt …)

Aliasing och referenser

- En av de mest förvirrande aspekterna av objekt (och en vanlig källa till buggar) är referenser
- Enkelt beskrivet så kan olika variabler referera till samma objekt
  - Om x och y refererar till samma objekt säger vi att y är ett alias för x (och vice versa)
- En anledning till förvirringen är att det inte fungerar så för enkla typer (som tal och booleans), där *är* variabeln sitt värde och två olika variabler kan inte referera till samma tal

Två snarlika program

- Det här belyser skillnaden mellan variabler med objekt och med heltal
- Båda programmen: Skapa original-skapa kopia-ändra kopia-skriv ut original

```
c1 = Circle(Point(15,15), 5)
c2 = c1

c2.move(10,0)
print(c1.getCenter())
```

Skriver ut (25,15)
c1 flyttades av c2.move!

```
x1 = 1.0
x2 = x1

x2 = x2 + 3
print(x1)
```

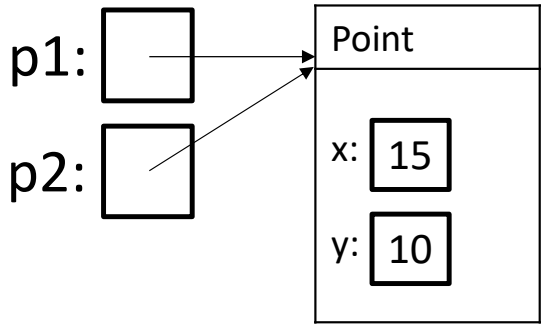
Skriver ut 1.0
x1 ändrades inte trots x2=x1

Notera: kodexplen här gör inget användbart, de visar bara skillnaden

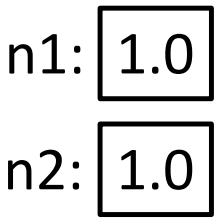
Vad innehåller egentligen en variabel?

- För enkla typer som tal och boolska värden innehåller variabeln själva värdet
  - Två heltalsvariabler kan ha identiska värden, men det är alltid två oberoende heltal, att ändra den ena ändrar inte den andra
- För objektstyper (inklusive arrayer) lagrar variabeln egentligen en referens till det underliggande objektet
  - Två variabler kan hänvisa till *samma* objekt (aliasing)

```
p1 = Point(15,10)
p2 = p1
```




```
n1 = 1.0
n2 = n1
```



p1 och p2 är alias

Hur vi trots det talar om variabler

- Oftast talar vi om variabler utan att bry oss om att de är referenser
- Vi säger ”c är en cirkel” eller ”arr är en array av heltal” istället för ”c är en referens till ett cirkelobjekt” osv.
- Oftast är skillnaden inte viktigt och det är bekvämt att inte bry sig om skillnaden mellan en variabel och objektet det hänvisar till

Andra källor till aliasing

- Förutom att satser i stil med c1=c2 skapar alias ifall c2 är en objektstyp uppstår de mer naturligt på ett par ställen:
  - I arrayer, till exempel [c1,c2,c1]
  - Vid funktionsanrop, varje gång vi använder ett objekt som parameter!

```
def someFunction(point):
    …

p = Point(0,0)
someFunction(p)
```

Parametern point blir ett alias för p i det här anropet

Ett klurigt exempel

- Studera det här programmet som gör två utskrifter, vad skrivs ut?

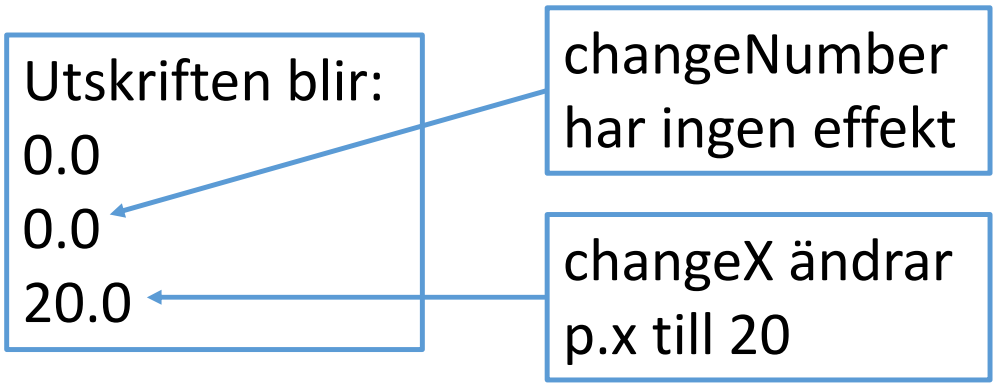
```
def changeNumber(n):
    n = 10.0

def changeX(point):
    point.x = 20.0

p = Point(0,0)
print(p.x)
changeNumber(p.x)
print(p.x)
changeX(p)
print(p.x)
```

Tar ett tal och ändrar det(?)

Tar ett objekt och ändrar dess x-attribut



Analys av klurigt exempel

```
def changeNumber(n):
    n = 10.0

def changeX(point):
    point.x = 20.0

p = Point(0,0)
changeNumber(p.x)
print(p.x)
changeX(p)
print(p.x)
```

Den här funktion gör aldrig något användbart!

Den här funktion ändrar ett attribut i ett objekt

Skickar en kopia av talet p.x till changeNumber

Skickar en referens till objektet p till changeX

Utskriften blir:
0.0
20.0

Ännu mer förvirring!

- Vad skriver det här programmet ut? (vad är x-positionen för p – 0 eller 5?)

```
def changePoint(point):
    point = Point(5,5)

p = Point(0,0)
changePoint(p)
print(p.getX())
```

point blir ett alias för p

… så det här skriver över p?

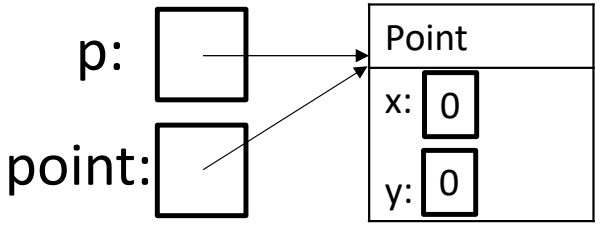
Rätt svar är 0!

Analys av föregående exempel

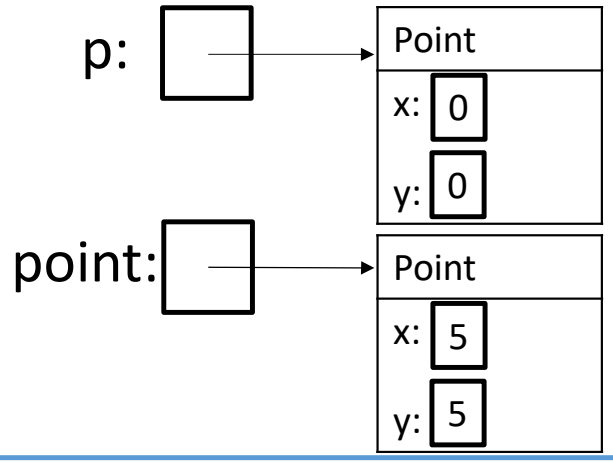
```
def changePoint(point):
    # Innan tilldelning
    point = Point(5,5)
    # Efter tilldelning

p = Point(0,0)
changePoint(p)
print(p.getX())
```

Innan tilldelningen (point = …):



Efter tilldelningen (point = …):



Tilldelningen `point = Point(5,5)` ändrar vad point refererar till!
Påverkar inte det p refererar till.
point slutar vara ett alias för p!

En bra tumregel

- Objekt skapas bara när en konstruktor anropas

```
def changeX(point):
    point.x = 20.0

p = Point(0,0)
changeX(p)
```

- Koden ovan skapar bara ett enda Point-objekt, point är ett alias för p
- Ibland kan så klart anropet till konstruerare vara undangömt i metoder, till exempel om vi skriver p = someFunction(2) så vet vi inte ifall p är ett nytt objekt skapat av someFunction eller ett alias för ett redan existerande objekt

Aliasing och icke-muterbara klasser

- En strängvariabel innehåller referenser, så skriver vi a=”hello” och b=a så finns det bara ett strängobjekt med två alias (a och b)
- Eftersom strängar inte är muterbara finns ingen observerbar skillnad mellan två variabler med olika identiska strängar och två alias för samma sträng
- Strängar beter sig i princip som tal och andra enkla typer, vi kan inte göra något med b som ändrar strängen i a
- Det här är en fördel med icke-muterbara klasser, aliasing ställer sällan till problem med klasser som inte är muterbara
  - Nackdelen är att vi inte kan mutera dem ens när vi vill
  - En funktion som changeStuffInString(x) kan inte ändra värdet på x
  - Vi får istället skriva x = transformIntoNewString(x) och returnera värdet

Grafik och grafikbibliotek

Grafik och grafiska gränssnitt

- De flesta program har något slags grafiskt gränssnitt
- För icke-programmerare så är det oftast gränssnittet som ”är programmet”, för programmerare är det snarare något man placerar ovanpå de viktiga delarna (objekt, funktioner osv. som representerar det programmet gör)

Design av grafiska gränssnitt

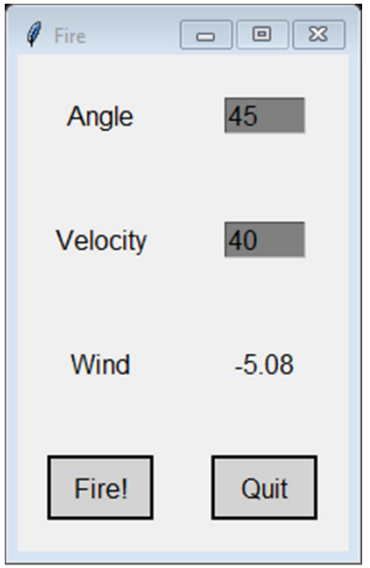
- De ”mjukare” delarna av att designa snygga och användarvänliga gränssnitt täcks inte av den här kursen
- Vi fokuserar på de tekniska aspekterna

Ett par viktiga principer

- Uppdelning av gränssnittet i olika grafiska komponenter
  - Eftersom vi inte kan rita direkt i programkod struktureras grafiska gränssnitt genom att kombinera olika komponenter
  - Varje komponent är ett objekt som representerar ett synligt grafiskt element till exempel ett fönster, en knapp, …
  - Vi beskriver gränssnittet med kod som skapar och manipulerar dessa komponenter
- Uppdelning av ett program i kod för grafik och kod för modell
  - Separera kod som enbart beskriver hur programmet ser ut från kod som beskriver vad programmet gör
  - Exempel: I ett spel vi inte blanda ihop kod som räknar ut vem som vinner med koden som ritar ut ett gratulationsfönster till användaren
  - Exempel: I ett kalkylprogram (typ MS Excel) vill vi inte blanda ihop hur värdet i en cell beräknas med val av teckensnitt eller liknande

Exempel, beskrivning av GUI

- Från labb 2
- Grafiken består av ett fönster
- Fönstret är uppdelat i ett 2x4-rutnät
- De första två raderna i rutnätet består var och en av en textkomponent ('angle' och 'velocity') och en inmatningsruta
- Den tredje raden består av två textkomponenter
- Den fjärde raden består av två knappar med texterna 'Fire!' Och 'Quit’
- När vi skriver koden för grafiken vill vi använda den här strukturen (skapa ett fönster, dela upp det i rutnät, …)



Grafikbibliotek

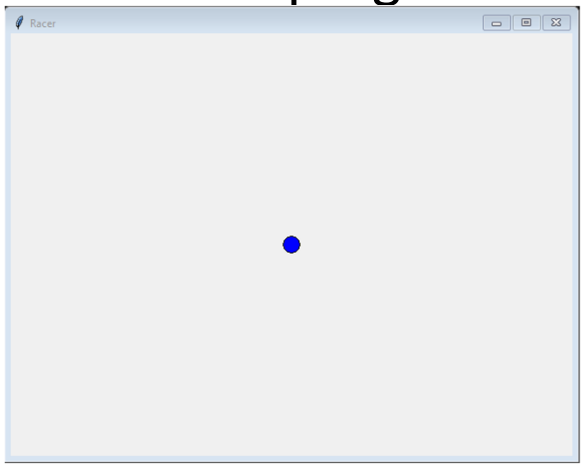
- I Python och de flesta andra större språk finns mängder av grafikbibliotek, specialiserade för specifika ändamål eller mer allmänna. Några kategorier av grafikbibliotek:
  - GUI-bibliotek för att skapa grafiska gränssnitt (graphics.py är baserat på GUI-biblioteket Tkinter)
  - Bildhantering och analys: Exempelvis för ansiktsigenkänning, OCR/textinläsning
  - Videohantering
  - 3D-motorer: Används för spel, simuleringar, avancerade visualiseringar osv.
  - Specialiserade bibliotek som Charts-bibliotek för att måla diagram

Graphics.py

- Det lilla grafikbiblioteket vi använder i den här kursen omfattar en handfull klasser för enkla geometriska figurer som streck och cirklar, och klasser för fönster figurerna kan placeras i
- Dels har det färre klasser och funktioner än ett ”riktigt” bibliotek skulle ha och dels undviker det några tekniker, exempelvis högre ordningens funktioner
- Vill ni istället lära er använda Tkiter på egen hand går det också bra

Bygg en racerbil

- Vi ska använda graphics.py för att bygga ett program där vi styr en bil (en blå cirkel) genom ett fönster. Så här ser programmet ut när det startas:



- Tangentbordets piltangenter accelererar bilen i fyra riktningar
- Mellanslag bromsen bilen
- Backspace startar om simuleringen

Vad behöver vi kunna?

- Utifrån uppgiftsbeskrivningen kan vi dela upp uppgiften i olika delar och sammanställa olika tekniska saker vi behöver kunna utföra
- Vi behöver kunna:
  - Skapa ett fönster
  - Rita en cirkel
  - Läsa av tangentbordstryckningar
  - Animera rörelse
- Allt vi behöver för det här finns i graphics.py, plus det vi redan kan om Python (loopar, if-satser osv.)

Klassen GraphWin

- En viktig klass i graphics.py är GraphWin
- Varje GraphWin objekt representerar ett fönster
- Till skillnad från exempelvis cirkelklassen så visas fönstret grafiskt direkt när det skapas med konstrueraren GraphWin

racer.py: Första steg

- Vi skapar en fil som heter racer.py
- Det första vi måste göra är att importera grafikbiblioteket

```
from graphics import *
```

- Om det här inte fungerar beror det troligen på att ni inte har graphics.py i samma katalog som racer.py

racer.py: Skapa fönstret vi kör i

- Vi börjar med att skapa fönstret

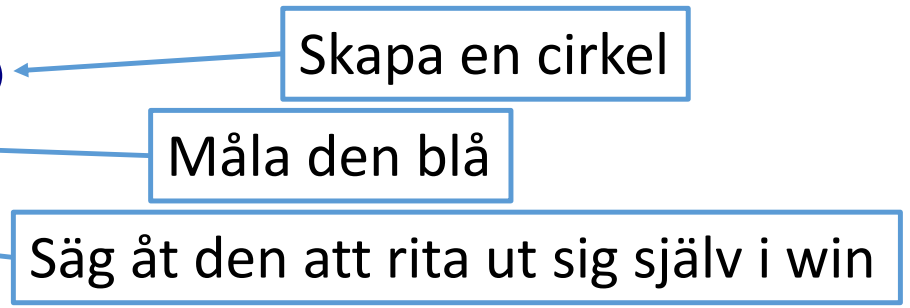
```
win = GraphWin("Racer" , 640, 480, autoflush=False)
win.setCoords(-320, -240, 320, 240)
```

- Här använder vi konstruktorn för klassen GraphWin
- 640*480 är x- och y- storleken på fönstret i pixlar
- win.setCoords ändrar det interna koordinatsystemet så att (0,0) är i mitten av fönstret (där bilen ska starta)
  - (-320,-240) är längst ner till vänster och (320,240) är längst upp till höger
- autoflush = False anger att fönstret bara ska uppdateras när man kör update(), se ”Controlling Display Updates” i dokumentationen för graphics.py
- Att köra den här koden ger oss ett helt tomt fönster

racer.py: Skapa ”bilen”

- Bilen är bara en blå cirkel
  - Inledningsvis ska den stå på (0,0) och
  - En radie på 10 pixlar är kanske lagom?

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



- Färgen sätts inte i konstruktorn utan genom ett anrop till setFill
- Slutligen säger vi åt racer att rita ut sig själv i fönstret win (som vi skapade på föregående slide)
- Att köra programmet med dessa rader räcker för att visa fönstret med en helt stillastående bil

Tre tänkbara sätt att rita en cirkel

- I graphics.py har varje grafisk komponent en metod som heter draw för att rita ut komponenten i ett angivet fönster

```
racer.draw(win)
```

- Ett annat designval hade kunnat vara att säga åt win att rita cirkeln

```
win.draw(racer)
```

Tänkbar alternativ kod, inte körbar!

- Eller en funktion istället för en metod (inte så objektorienterat)

```
draw(win, racer)
```

Tänkbar alternativ kod, inte körbar!

- Alla tre kör en funktion/metod som har tillgång till win och racer. Vilket som är bäst är en smakfråga, men den första är den som används i graphics.py

racer.py: Animationsloopen

- Resten av programmet är en animationsloop
- Kör ”för alltid”
- Sista steget i varje varv är att måla om grafiken med en metod från graphics.py

```
animationSpeed = 50
while(True):

    update(animationSpeed)
```

Det här värdet anger antal animationssteg per sekund

Oändlig loop (tills programmet stängs)

Här lägger vi all kod som ska köras varje animationssteg

Målar om all grafik, väntar ifall det gått för kort tid

update() i graphics.py

- Update gör två saker:
  - Får grafiken att målas om (när autoflush=False har angetts i fönstret)
  - Reglerar hastigheten på en animations-loop
    - För att det ska fungera är det viktigt att inget annat i loopen ”väntar” på något, till exempel användarinmatning annars stannar animationen

```
while(True):
    …
    update(10)
```

Den här loopen körs som mest 10 gånger per sekund

racer.py: Få bilen att röra sig

- Vi lägger till två variabler (utanför loopen) som anger x- och y-hastighet (negativa värden betyder vänster/ner, positiva höger/upp)
- Varje varv av loopen flyttas racer (cirkeln) baserat på hastigheten

```
xvelocity = 0
yvelocity = 0
while(True):

    racer.move(xvelocity,yvelocity)

    update(animationSpeed)
```

Mer saker måste hända här!

- Eftersom värdena är 0 kommer den stå still
  - Ändrar vi ena/båda till 1 eller -1 och provkör kommer bilen rulla iväg

Tangentbordsinmatning

- Det finns två metoder för tangentbordsinmatning i GraphWin

`getKey()` Pauses for the user to type a key on the keyboard and returns a string representing the key that was pressed.

Example: `keyString = win.getKey()`

`checkKey()` Similar to `getKey`, but does not pause for the user to press a key. Returns the last key that was pressed or `""` if no key was pressed since the previous call to `checkKey` or `getKey`. This is particularly useful for controlling simple animation loops (see Chapter 8).

Example: `keyString = win.checkKey()`
Note: `keyString` may be the empty string `""`

- Båda ger en sträng som resultat, skillnaden är att den första väntar tills användaren tryckt på en tangent, medan den andra ger senaste tangenten som tryckts sedan förra anropet
  - Eftersom vår animation måste fortsätta köra använder vi checkKey
- Notera att båda bara fungerar ifall fönstret win är i fokus (så se till att du växlat till fönstret och inte skriver i din editor eller liknande!)

racer.py: Inmatningsdelen

- I varje steg av loopen behöver vi kontrollera om användaren trycker på någon tangent, och justera hastigheten

```
xvelocity = 0
yvelocity = 0
while(True):
    key = win.checkKey()
    if key == "Up":
        yvelocity += 1
    elif key == "Down":
        yvelocity -= 1
    elif key == "Right":
        xvelocity += 1
    elif key == "Left":
        xvelocity -= 1
    ...
```

Ger en sträng för vilken tangent användaren trycker på ("" ifall ingen tangent)

Ökar/minskar hastigheten med 1 (yvelocity += 1 är en kortform för yvelocity = yvelocity + 1)

racer.py: Nu kan vi köra bilen!

- Algoritmen för huvudloopen vi har just nu kan beskrivas så här:
  - Inmatningsdel, justera hastighetsvärden baserat på tangenttryck
  - Flytta bilen baserad på nuvarande hastighetsvärden
  - Vänta tills det gått 1/50 sekund och måla om grafiken
  - Upprepa

```
while(True):
    key = win.checkKey()
    if key == "Up":
        yvelocity += 1
    elif key == "Down":
        yvelocity -= 1
    elif key == "Right":
        xvelocity += 1
    elif key == "Left":
        xvelocity -= 1

    racer.move(xvelocity,yvelocity)

    update(animationSpeed)
```

racer.py: Implementera broms

- Vi vill att om användern håller inne mellanslag så ska bilen bromsa in
- Ett sätt att få en ”mjuk” inbromsning är att minska hastigheten med en till exempel 10% varje animationssteg:

```
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
...
```

Fördel: Fungerar för både negativa och positiva hastigheter

Tips: Arbeta inkrementellt

- Ett vanligt nybörjarmisstag är att skriva ett helt program, ibland med flera hundra rader kod, och sen försöka köra det
  - Programmet kommer oundvikligen innehålla massor av buggar
  - Det är en mardröm att felsöka för buggarna döljer varandra
  - Ofta samma misstag på flera ställen
- Rätt sätt: Börja med ett program som gör *någonting*, men inte *allting*
  - Provkör, fixa buggar och utöka sedan med ett par rader ytterligare kod, testkör den osv.
  - När du stöter på en bugg vet du var du nyss har ändrat och alltså var buggen uppstår

racer.py: starta om animationen

- När användaren trycker backspace ska bilen återgå till startpositionen
- Problem: racer.move tar relativa värden, hur gör vi för att flytta den till (0,0)?
  - Exempel: Om bilen står på (-23,30) så ska vi köra racer.move(23,-30)
  - Vi kan få ut positionen (-23,30) med racer.getCenter(), som ett Point-objekt
    - racer.getCenter().getX() ger -23
    - racer.getCenter().getY() ger 30
- Slutsats: Följande sats flyttar en cirkel (racer) till (0,0):

```
racer.move(-racer.getCenter().getX(), -racer.getCenter().getY())
```

- Det här är ett bra exempel på ett invecklat uttryck där punktnotation staplas i flera nivåer, stirra på det tills ni förstår hur det fungerar!
  - En oerfaren programmerare skulle hjälpas av att definiera variabler för nuvarande x- och y-positioner och köra racer.move(-xpos, -ypos)

racer.py: Starta om (forstättning)

- Vi startar om genom att flytta och stanna bilen

```
    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
    …
```

racer.py: Lägg till en maxhastighet

- Bilen kan röra sig väldigt snabbt, låt oss lägga in en hastighetsgräns
- Ovanför loopen:

```
speedLimit = 5 # How fast is the racer allowed to move
```

- Inuti loopen (efter inmatningsdelen)

```
yvelocity = min(speedLimit, yvelocity)
yvelocity = max(-speedLimit, yvelocity)
xvelocity = min(speedLimit, xvelocity)
xvelocity = max(-speedLimit, xvelocity)
```

- Funktionerna min och max ger det lägsta/högsta av dess parametrar
- De fyra raderna motsvarar för hög hastighet upp/ner/höger/vänster
- Exempel: om yvelocity > speedLimit sätter första raden yvelocity=speedLimit, annars yvelocity = yvelocity (som int har någon effekt)
- Man hade kunnat åstadkomma samma effekt med flera if-satser

racer.py: Teknisk analys

- En bra vana för era första program är att studera dem och beskriv de olika komponenterna med begreppen ni lärt er i kursen
- Beskrivningen är teknisk på så sätt att en programmerare kan utföra den utan att veta något om vad koden egentligen gör (ritar och styr en bil)

racer.py …

- Skapar tre objekt med konstruerare för klasserna GraphWin, Point och Circle:

```
win = GraphWin("Racer" , 640, 480, autoflush=False)
racer = Circle(Point(0,0), 10)
```

- Använder metoderna `setCoords()` och `checkKey()` från klassen GraphWin
- Använder metoderna setFill, draw, move och getCenter från klassen Circle
- Använder metoderna getX och getY från klassen Point (måste veta att getCenter ger en Point!)
- Aliasing sker exempelvis i racer.draw(win), där draw-metoden får en referens till vårt fönster, så cirkeln kan ritas i fönstret som redan finns (utan aliasing skulle det inte fungera!)
- Mutation sker exempelvis i setFill, move och draw (alla metoder som har någon effekt)

racer.py: Provkör!

- Om du inte vill skriva din egen kopia av racer.py finns en färdig kopia på kurshemsidan

Ur boken

Föreläsningen

- Kapitel 4 i kursboken
- Kapitel 8.6 i kursboken

QA-sessionen

- Öva på att använda graphics.py för att skapa fönster och rita ut grafiska komponenter
- Bekanta er med dokumentationen för graphics.py och hur man lär sig använda en klass genom att studera dess konstruerare och metoder