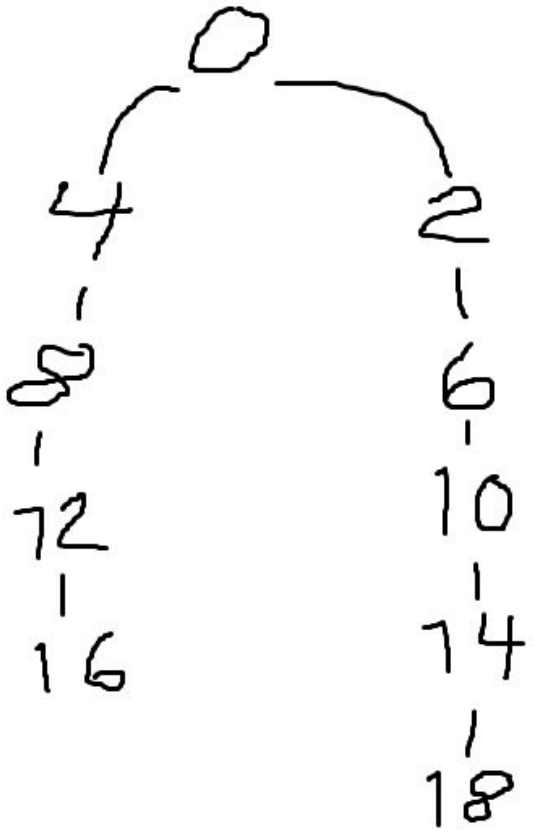
1).

Class Tree: definerar en klass som heter “Tree”. När du skapar ett obejekt med klassen “Tree” så kommer det första argumentet som du skickar in vara “Data” som sätts till obejektet. Objektet kommer också ha egenskaperna left och right som båda har värdet “None”. Om du printar ditt obejekt så kommer du få det som står under “def __str__ (self)” alltså värdet på self.data.

När programet först startar så ger den root värdet “None”, sen används en for loop för att kalla på funktionen insert 10 gånger. Värdet på data kommer va olika alla gånger som loopen körs: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18. n kommer börja på 0 och sen gå upp ett nummer i taget tills 9.

Sen gör insertfunktionen om dessa värden till “Tree” objekt. Om N är jämmt så kommer Trädobejektet att lägas in på den vänstra sidan av trädet. Om det är ett udda tal så kommer det att läggas in på den högra sidan.

Eftersom node = None första gången du kör så kommer 0 vara i mitten(Roten) av trädet.
om man visualiserar trädet så ser det ut så här



Om du kör programet så kommer den att printa ut det Högra ledet nerifrån upp, sedan det vänstraledet uppifrån ner.

18 14 10 6 2 0 4 8 12 16

2).

När progament startar så kommer det att kalla på Funkis med argumentet “Hello”.

Det här är vad som kommer att printas

Hello	- Första funkis	1
llo	- Andra funkis	3
o	- Tredje funkis	5
lo	- Andra no_funkis	4
Ello	- Första no_funkis	2

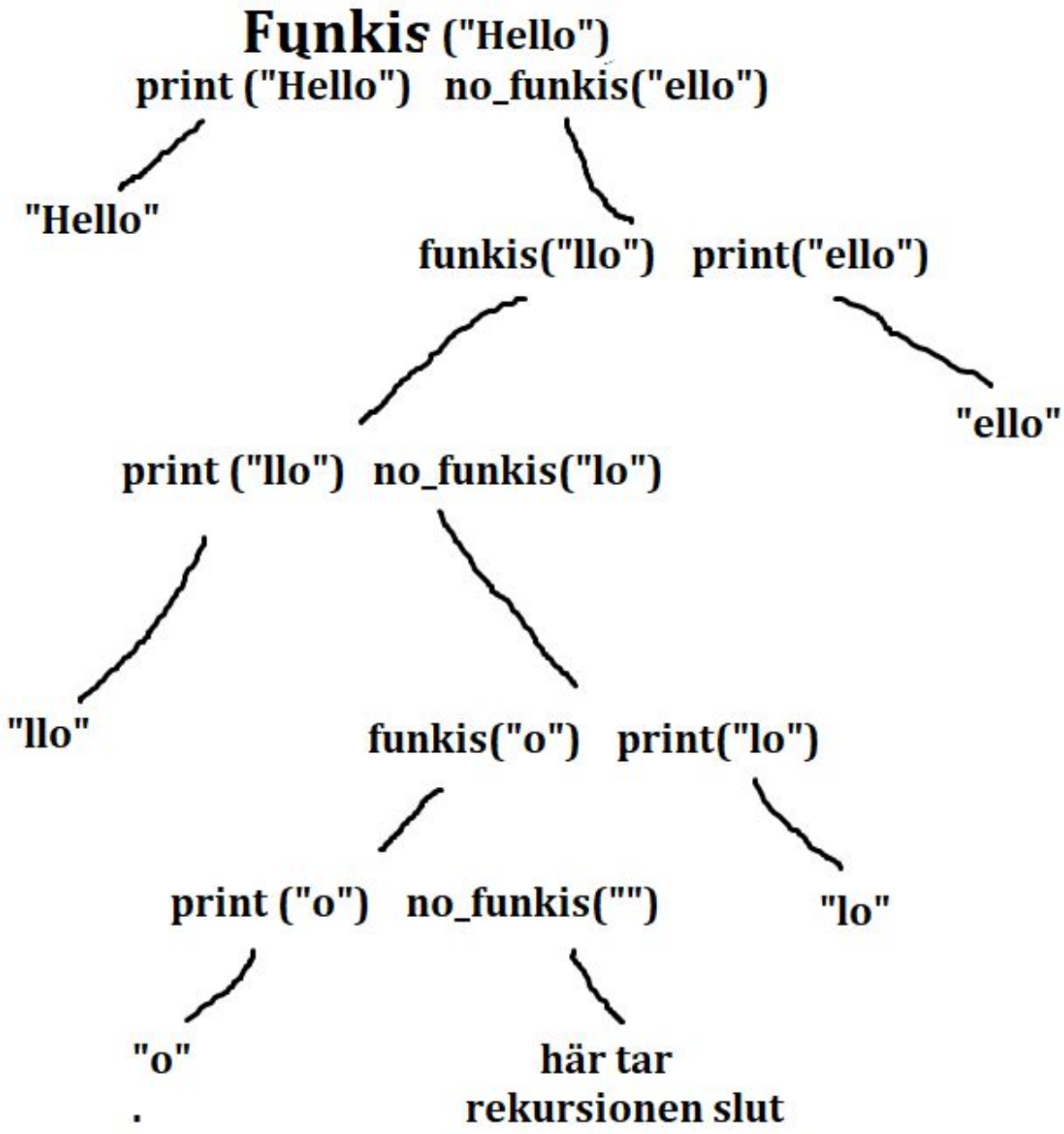
Eftersom no_funkis printar sin string efter att den gör sin rekursiva funktionskallning så betyder det att, det första värdet som no_funk har “ello” är det som kommer printas sist.

Ello	- första funkis	2
Lo	- andra funkis	4
O	- tredje no_funkis	5
Llo	- andra Funkis	3
Hello	- första no_funkis	1

Siffrorna visar i vilken ordning som funktionerna blir kallade, trots att “Hello” med alla boxtäver är det första som kallas på när man kör no_funkis(“Hello”) så är det det sista som printas.

Det här trädet är en visualisering av hur rekursionen går till när man kallar på funkis(“Hello”)

Saker händer från vänster till höger



3)

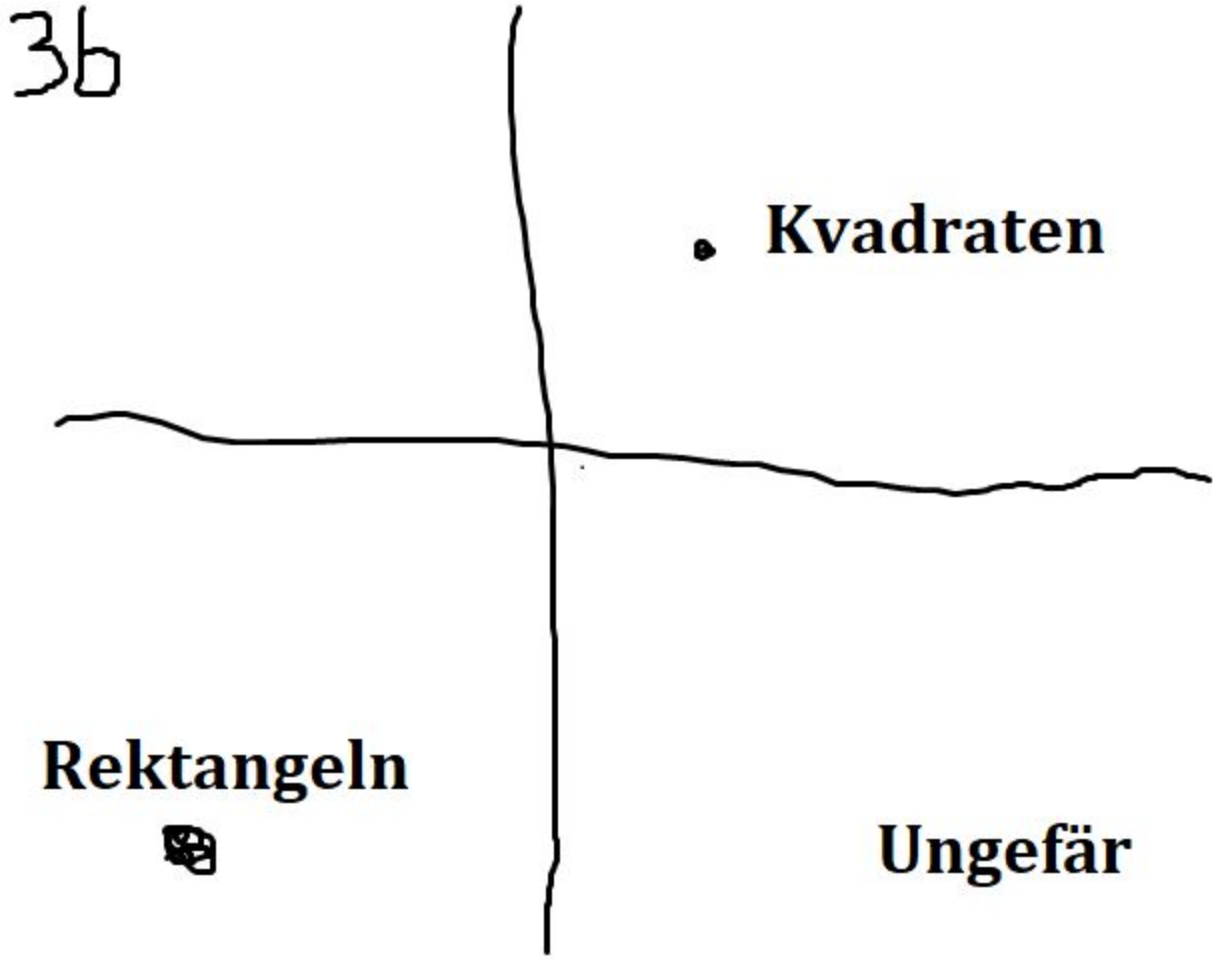
a)

import turtle

```
window = turtle.screen()

pen = turtle.Turle()

pen.penup()

pen.goto(55, 55)


pen.forward(20)

pen.right(90)

pen.forward(20)

pen.right(90)

pen.forward(20)

pen.right(90)

pen.forward(20)

pen.right(90)


pen.goto(-75, -75)

pen.forward(30)

pen.right(90)

pen.forward(20)

pen.right(90)

pen.forward(30)

pen.right(90)

pen.forward(20)
```

pen.right(90)

b)



c)
Turtlegrafik är väldigt användtbart om du är ny till programering och vill lära dig att använda lite grafik. Det är en av de enklaste typer av grafik du kan använda och den finns redan på python så du behöver inte ladda ner något vilket gör det till en bra introduktion. Om du vill visualisera din kod så är det bra att ha någon sorts grafik så att du se saker. Programmering med grafik blir också mycket mer underhållande för många vilket kan hjälpa att hålla intresset och motivationen uppe.

4)

a) select * from Player where Racket = Ficher;

b) select * from Player, Tour where Racket = Adidas and Place = 1;

c) select * from Player where Name not in(select Name from Tour);

d)

Name	Shoe
Agassi	Nike
Roddick	Keds

5)

a) Att något är cross-platform betyder att det är väldigt flexibelt och kommer att funka bra på olika platformar, istället för att vara specialdesignat till en specifik platform

b) ett native UI betyder att grafiken använder resurser som redan finns på datorn. Det betyder att om du använder en Mac så kommer det se ut som ett Mac program, om du använder en PC kommer det se ut som ett windows program.

c + d)

Många program tjänar på att ha nativa UI, då det kommer passa in bättre på plattformen som du använder. Men om du har designat något som du vill ska se ut på ett väldigt specifikt sätt, och du vill att det ska se ut så på alla plattformar så är det bra att använda ett icke Native UI. Om du vill göra ett program som spotify kanske det är bättre

6)

a)

Det som gör databaser så mycket bättre än vanliga sätt att spara information är relationerna som den använder. Om du har stora mängder information så kan du söka i databasens tabeller med hjälp av relationerna mellan informationen. Om det är små mängder med information så är det kanske inte nödvändigt att använda Databaser, men om det handlar om mycket information så är mycket mer effektivt att ha Databaser.

b)

i - Eftersom böcker oftast bara har 1 huvudförfattare, men en författare kan skriva många böcker så är det nog smart att använda en one-to-many relation.

ii - Varje användare har bara ett personnummer, och om allting är rätt så ska varje personnummer bara tillhöra en person, så det borde räcka med en one-to-one relation här

iii - Precis som med person nummer så borde de flesta bara ha en mail adress per användare, så det borde räcka med en one-to-one här. men om du vill att användare ska kunna koppla ihop flera mailadresser till sin användare så kan du också använda one-to-many.

7)
Om du gör något som du tror kanske kan få programet att kracha så är det smart att använda en try sats.

```
try:
    print (“Exempelkod”)

except XXX:
```

Efter en try sats så måste du ha en except sats. Det är vad som händer om try satsen inte funkar(programet hadde krachat i vanliga fall). där jag skrev XXX skriver du vilken typ av Error som du förväntar dig. några exempel är ValueError(när värden är av fel typ), DivisionbyZeroError(När den försöker dela något på 0), Indexerror(när man försöker kalla på index som inte finns).

8)

```
Class Product():
    def __init__(self, name, stock, price):
        self.name = name
        self.stock = stock
        self.price = price

    def getinfo(self):
        print (“Produktnam:”, self.name, “\nLagersaldo:”, self.stock, “\nPris:”, self.price, “kr”)

    def newstock(self, stock):
        self.stock = stock

    def newprice(self, price):
        self.price = price
```

prod01 = Product(“Dator”, 1000, 5000)

prod01.newstock(3)

prod01.newprice(12000)

Fördelar som finns med att andvända objektorientering är att alla värden sparas till obejektet. När man jobbar med data och information så är det jätte användbart att kunna klassifiera saker, så att man använder Klasser mycket när man programerar är inte så konstigt.