1. Träd

Objektet Tree() har instansvariablerna letter, number, left och right.

Letter lagrar bokstaven som kommer in från strängen.

Number är för att räkna hur många gånger bokstaven förekommer i strängen.

Left och right är platser för nya objekt att läggas till som barn till objektet.

Koden fungerar enligt följande:

Första bokstaven i strängen a_string ”f” kommer in i insert(), kollar om node == None dvs om trädets första nod är tom. Eftersom trädet är tomt läggs bokstaven f i variabeln letter och objektet läggs som rot i trädet.

För nästa bokstav i stängen: l, kollar koden om node == None – vilket den inte är.

Koden går vidare till nästa ifsats som kollar om bokstaven är lika med bokstaven som ligger i variabeln letter på trädets node. Eftersom ”l” == ”f” är Falskt går koden vidare till nästa if-sats

Koden kollar där om bokstaven ”l” är mindre än bokstaven som ligger under node.letter, om den är mindre läggs den till vänster i trädet, om den är större läggs den till höger i trädet. Eftersom bokstaven l är större än bokstaven f skapas ett nytt objekt med instansvariabeln letter = l, som läggs i den högra noden på det tidigare objektet (med letter = f).

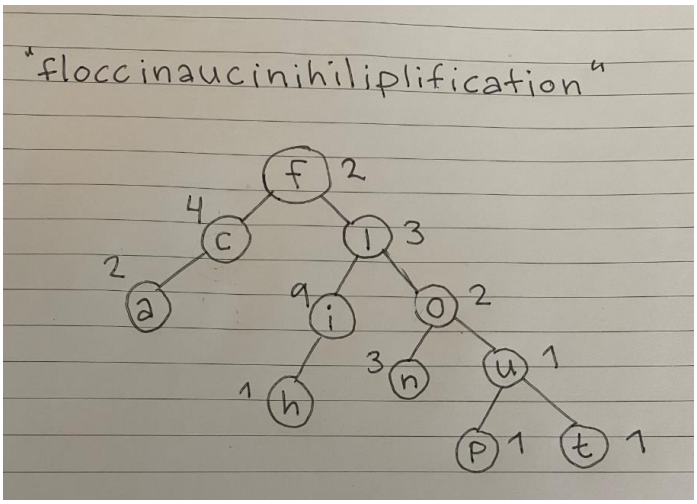
Nästa bokstav o går igenom samma procedur och läggs till höger om objektet f, men där ligger redan objektet l så objekt o fortsätter vandra ner och läggs på höger nod på objektet l, då o är större än l.

När nästa bokstav c kommer in så görs samma procedur och eftersom c är mindre än f så läggs ett nytt objekt med letter = c i den vänstra noden på objektet med letter=f.

När nästa bokstav som även denna är c kommer in så går koden in i den andra ifsatsen (if data == node.letter) då denna blir True efter som c == c är sant. Här adderas variabeln number med 1 på objektet med letter = c, vilket räknar hur många gånger bokstaven förekommer.

Koden fortsätter på detta sätt genom hela strängen. Om bokstaven redan finns i ett objekt i trädet så adderas räknaren number med 1 och om bokstaven inte finns så läggs den antingen till höger eller vänster ned i trädet beroende på om den är större eller mindre än noderna i trädet. Bokstaven vandrar ner i trädet tills den hittar en tom nod och det nya trädobjektet placeras där.

Bilden visar hur noderna läggs in i trädet och siffran på sidan visar hur många gånger ordet förekommer.



2. Rekursion

```
def count_odd(l):
    if len(l) > 0: # Basecase - Om längden på listan är 0 skall rekursionen av
slutas genom return 0
        if l[0] % 2 != 0: # Kollar om första talet i listan är ojämnt
            return l[0] + count_odd(l[1:]) # Om det är ojämnt adderas talet me
d count_odd() med resten av listan som parameter
        else:
            return count_odd(l[1:]) # Om talet är jämnt går vi vidare och till
kallar funktionen count_odd() igen med resten av listan som parameter
    return 0

a_list = [4,7,88,3,3,66,101,43,56]
print(count_odd(a_list)) ##bör skriva ut 157 på skärmen
```

3. SQL och Python

```
import sqlite3
conn = sqlite3.connect('filmer.db')
cursor = conn.cursor()
cursor.execute("SELECT film FROM Nya_Filmer WHERE Minuter < 110;")

rows = cursor.fetchall()

print("Filmer")
for row in rows:
    for i in row:
        print(i)

conn.close()
```

4. Utveckling: debugging, testning och stegvis förfining

a) Genom att använda en debugger kan man sätta ut breakpoint och på så sätt pausa koden där man vill kika på vad som händer i detalj. Ett annat sätt är att se koden som byggstenar där man går igenom bit för bit. Det kan vara att fokusera på endast en funktion tills den fungerar som det ska och sen gå vidare till nästa. Man kan även skriva pseudokod som en strategi för att innan man skriver själva koden konceptuellt gå igenom vad programmet skall innehålla och skriva upp detta. Man kan t.ex skriva upp de funktioner som skall finnas och des parametrar samt vad de skall returneras.

b) Genom stegvis förfining kan man undvika tidiga fel i koden som man annars hade haft svårt att hitta senare. Man undviker att bygga fel från början. Det kan även förebygga missförstånd om man är flera som kodar tillsammans då det ger en tydlig blid av vad som faktiskt händer/ska hända i programmet man skriver.

c)

def rita_kvadrat(kvadratens längd):

for i in range(4):

kvadrat = rita kvadratens längd

return kvadrat

for i in range(n)

rita_kvadrat(20)

gå_vänster(360/n)

5) Turtle

a)

```
import turtle

t = turtle.Turtle()

def square(lenght, t):

    for i in range(4):
        s = t.left(90)
        s = t.forward(lenght)
    return s
n = 8

for i in range(n):
    square(30, t)
    t.left(360/n)


turtle.done()
```

b)

```
import turtle

t = turtle.Turtle()

t.pensize(5)
t.speed(0)
t.color("sienna4")

t.left(90)
t.forward(200)
t.right(90)
t.color("red")
t.begin_fill()
t.circle(100)
t.end_fill()
t.pensize(3)
t.color("white")
t.penup()
t.left(90)
t.forward(100)
t.pendown()

for i in range(120):
    t.forward(2+i/4)
    t.left(30-i/12)


turtle.done()
```

6. Matplotlib

a)

Bra – Två tydligt olika färger är valda, vilket ger en bra tydlighet i visualizeringen rent färgmässigt.

Bra – Man har märkt upp namnen på iris-sorterna i visualiseringen.

Dåligt – Det står inte vad siffrorna på y eller z axeln representerar, vilket gör det svårt att avläsa någon information från visualiseringen.

Dåligt – Diagrammet har ingen titel vilket också gör det svårt att veta vad visualiseringen handlar om.

b)

Det är viktigt att tänka på att inte välja för lika färger till olika saker som skall visualiseras. Det är även viktigt att ha i åtanke att vi människor associerar olika färger med olika saker, t.ex rött – fara, grönt – okej. (Rödljus). Därför är det bra att vid val av färg tänka på vad man vill förmedla i visualiseringen så att man inte skickar felaktiga signaler via färgerna.

c)

Man kan ändra opaciteten (alpha) på prickarna så att man ser igenom dem lite och de får en kant, då flyter inte ihop. Man kan även ändra på längden av y-axeln så att man får ett större eller mindre fält. Riskerna med att manipulera scatterplottet på detta sätt är att datan kan bli vilseledande. Därför måste man ha detta i åtanke och göra det på ett tydligt och ärligt sätt för användaren.

7. Exceptions

a) Sysargs innehåller [prog1, 42, 16] – programmets namn ligger alltid på [0] platsen i listan och parametrarna från terminalen läggs in därefter.

b)

```
def check_if_int(value):

    try:
        int(value)
        return True
    except ValueError:
        print("Det går ej att konvertera till en int")
        return False

check_if_int("Sträng")
```

c)

```
def check_if_int(value):
    try:
        float_value = float(value)
        int(float_value)
        print("Talet har trunkerats till närmaste heltal")
        return True
    except ValueError:
        print("Det går ej att konvertera till en int")
        return False

check_if_int("3.0")
```

8. Objektorientering

a) Det är bra att använda sig av objektorientering då det gör att man kan återanvända kod. Som i exempelkoden där vi vill skriva ett program med anteckningsböcker. Om vi skapar objekt av classen Note (anteckningsbok) så kan vi återanvända den koden för att skapa flera anteckningsböcker utan att behöva definiera en anteckningsbok för varje gång.

Även när vi vill definiera innehållet i en anteckning, så kan vi använda oss av objektorientering för att lägga till innehåll i anteckningen.

b)

my_note = Note() – Eftersom objektet Note() behöver parameter titel för att skapas så kan inte objektet skapas.

my_note = Note("En ny anteckning") – Här har man lagt in en sträng i parametern titeln som är den enda parameter som behövs för att skapa en instans av objektet Note(), objektet kan därför skapas.

c)

För att skapa flera objekt med denna class gör man på följande sätt

anteckning1 = Note(”Inköpslista”)

anteckning2 = Note(”kom ihåg-lista”)

anteckning3 = Note(”föreläsning 2”)

Vill man skapa nya typer av anteckningar kan man skapa nya klasser som ärver av klassen Note. T.ex SchoolNotes() skulle kunna ärva av classen Note().

d)

```
class Note:
    def __init__(self, title=""):
        self.title = title
        self.checked = False

    def set_content(self, content):
        self.content = content
    def get_note(self):
        full_note = self.title.upper() + "\n" + self.content
        return full_note
    def set_checked(self):
        self.checked = True
    def get_checked(self):
        return self.checked
```

9. Numpy

a) NumPy är väldigt användbar för att bygga neurala nätverk.

b) Med NumPy-Arrayer kan man utföra matematiska operationer, vilket inte går att med vanliga python-listor.

b) Man kan i NumPy hantera 3D-arrayer vilket gör att man kan analysera bilder i flera olika lager. Detta används i AI-träning där man kan få AI-datorer att identifiera bilder via dessa lager av matriser.

10. GUI och grafik

a) En fördel är att det är lätt att använda om man skall göra enkla UI:n, det används som standard-grafik-modul i python , en nackdel är att utseendet inte är så uppdaterat och ser rätt gammalt ut.

b) Om man vill att layouten ska se lika dan ut på alla plattformar så är det bättre att göra en egen design, vill man bygga ett stort program så finns det mer hjälp att få om man väljer native då man inte behöver lägga ner lika mycket tid på själva designen eftersom grunder i programmets utseende (som knappar ex) kommer bestämmas av operativsystemet man kör programmet på.

c) Turtle är bra att använda i utbildningssituationer och när man vill ha en väldigt enkel design. Det går även bra att använda om man vill rita upp matematiska figurer.

Kivy är bra att använda om man vill göra mobilapplikationer och få den ”senaste” layoten på sitt program då kivy modulen är ny. Kivy baseras på googles Material Design vilket passar utmärkt i mobilapplikationer där man med fördel jobbar mindre med text och mer med ikoner.