

Djurparken

Innehållsförteckning

Introduktion.....	2
Förberedelser och Verktyg.....	3
Mjukvara och Verktyg.....	3
Mappstruktur.....	4
Mappar.....	5
Virtuell miljö.....	7
PyCharm.....	8
Git.....	13
Dokumentation och Requiriements.....	19
Implementering och Kod beskrivning.....	20
Introduktion.....	20
Filstruktur.....	21
Pytest.....	23
Kodbeskrivning: Av de olika filerna i programmet.....	25
Main.py(huvudfilen).....	26
Zoo.py.....	27
Animal.py.....	30
Animal sub-klasser.....	31
visitor.py.....	33
gui.py.....	35
Arbetsprocess och utvecklingsprocess.....	36
Utvecklingsprocess.....	37
Animal-klassen.....	41
Sub-klass Lion.....	44
Första testet.....	46
Utveckling av fler Subklasser: Giraffe, Elephant och LionCub.....	49
Automatiserade tester med pytest.....	53
Vad gör testen?.....	57
Användargränssnittet.....	59

Introduktion

Denna dokumentation beskriver utvecklingsprocessen och det färdiga programmet för Djurparken, ett interaktivt system där användare kan utforska och interagera med djur på en digital plattform. Syftet med projektet var att skapa ett användarvänligt program som simulerar upplevelsen av att besöka en djurpark. Programmet är designat för att erbjuda flera funktioner, såsom att visa information om djur, mata dem, och hantera interaktioner med olika djurarter.

Projektet är utvecklat med ett fokus på objektorienterad programmering, vilket innebär att klasser och metoder har använts för att representera zoo, djur och besökare. Genom en strukturerad design med UML-diagram och wireframes har programmet formats för att uppfylla en tydlig kravspecifikation. Funktionaliteten inkluderar att visa öppettider, söka efter djur, och hantera tillval som besökare kan köpa för att förbättra sin upplevelse.

Denna dokumentation syftar till att förmedla hur projektet har byggts upp från grunden, från design och planering till implementering och testning. Den innehåller kodexempel, skärmdumpar, och en reflektion över programmets styrkor och potentiella förbättringsområden. Genom att följa denna dokumentation får läsaren en detaljerad inblick i både den tekniska utvecklingen och de designmässiga valen som ligger bakom programmet.

Förberedelser och Verktyg

Mjukvara och Verktyg

Här är dokumentationen till vilken mjukvara jag använde samt olika verktyg för projektet.

Utvecklingsmiljö (IDE):

- PyCharm.(Windows)
- Atom. (macOS)
- Terminalen (Vet att detta inte är en utvecklingsmiljö. Använder dock denna på mac och linux för att köra script.)

Programspråk:

- Python 3.

Designverktyg:

- Lucidchart
- Figma.

Operativsystem:

- Windows/Linux/macOS.
Kommer använda mig av olika OS men mestadels Windows och macOS. Har även Linux mint. Men detta är sällan använt. Det kan skilja sig på printscreens på grund av detta.

Andra verktyg/mjukvara:

- Git
Kommer använda mig av versionshanterings programmet git. Använder mig även av git bash på Windows istället för cmd. Går ej in på detalj vad detta program gör. Läger med en länk. <https://git-scm.com/docs/user-manual.html>
- Github
För att skicka runt min kod mellan datorer.

Mappstruktur

Projektets mappstruktur är uppdelad enligt följande:

Djurpark/	#Rotmappen för hela projektet.
src/	#Källkod för programmet.
main.py	#Startfil för programmet.
gui.py	#GUI-komponenter och logik. (För grafiskt gränssnitt).
zoo/	#Paket för zoo-logik.
__init__.py	#Initierar zoo-paketet.
zoo.py	#Zoo-klassen.
visitors.py	#Besökare-klassen.
animals/	#Packet för klasser.
__init__.py	#Initierar Animals paketet
animal.py	#Bas-klass för djur.
lion.py	#Sub-klass för lejon.
lioncub.py	#Sub-klass av lejon.
giraffe.py	#Sub-klass för giraffer.
elepehant.py	#Sub-klass för elefanter.
tests/	#Tester för projektet.
test_zoo.py	#Test för Zoo-klass.
test_zoo.py	#Test för funktioner för programmet.
Assets/	#Resurser.
images/	#Bilder
docs/	#Doumentation för projektet.
dokumentation.odt	#Doumentation för projektet.
Requirements.txt	#Lista på olika moduler som behövs.

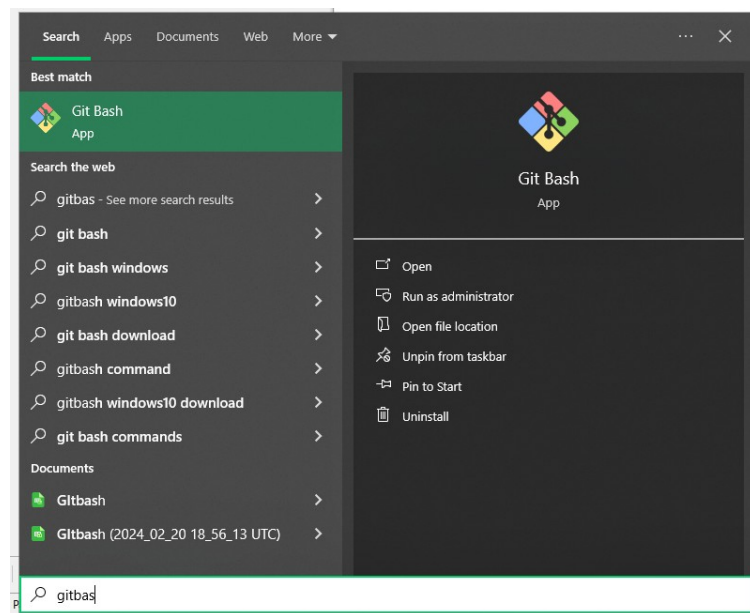
Mappar

Jag kommer att skapa mappar och filer lite eftersom. Men visar grunden av mapparna här.

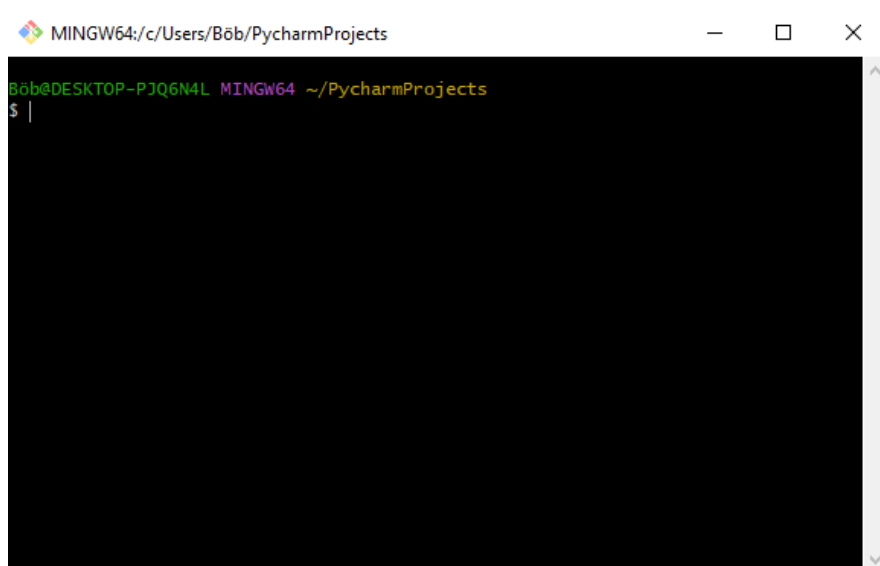
OBS: Du kan självklart skapa mappar/filer i GUI också om du tycker det är lättare. Även kommandona fungerar båda på Linux Terminal (bash) och macOS.

Kommer använda mig av Git bash som Shell. Alla kommandon som finns med är i **fetstil**.

Starta Git bash som admin.



Gå till mappen där du vill ha projektet. Med kommandot **cd**. Där du skriver i terminalen **cd Din/mapp/som/projekt/ska/ligga**



Sen i denna mappen ska du skriva in kommandona nedan i ordning.

Detta kommer att skapa root-mappen och undermappar som ska finnas. Kommer göra filerna som ska finnas allt eftersom.

mkdir Djurparken

cd Djurparken

mkdir src tests assets docs

cd src

mkdir zoo animals

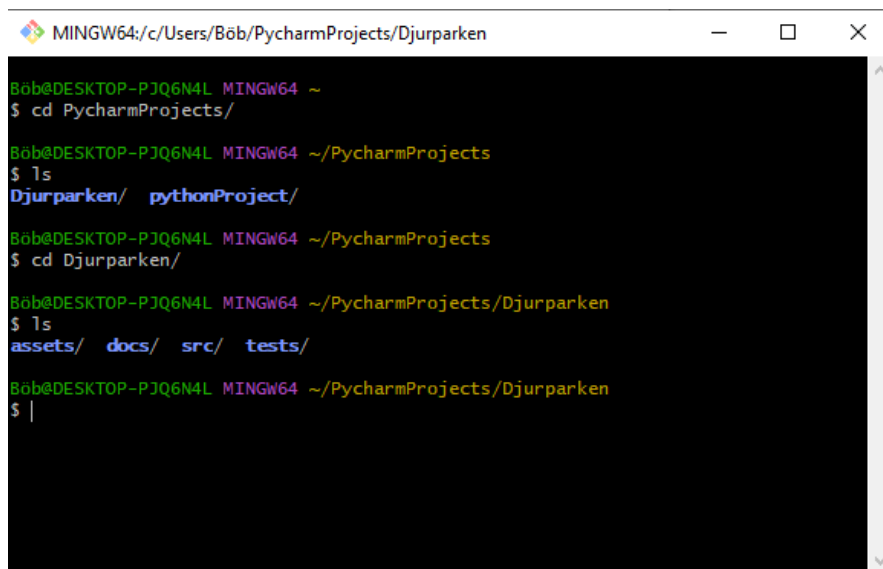
Detta kommer skapa alla mappar som ska finnas i projektet. Vi kan även skapa filerna men vi väntar med att göra det till senare i dokumentationen.

Virtuell miljö

I detta avsnitt beskriver jag hur jag skapade en virtuell miljö. En virtuell miljö innebär att man skapar en isolerad mapp där Python-bibliotek och interpretatorn installeras separat. Detta gör att man kan köra Python-filer utan att riskera konflikter med andra projekt eller systeminställningar. Genom att använda en virtuell miljö säkerställer man att varje projekt har sina egna specifika beroenden och versioner av bibliotek, vilket minskar risken för kompatibilitetsproblem.

Öppna en terminal. I Windows-miljö så används git-bash shell.

Navigera till din root mapp som du skapade tidigare.



```
MINGW64:/c/Users/Böb/PycharmProjects/Djurparken
Böb@DESKTOP-PJQ6N4L MINGW64 ~
$ cd PycharmProjects/

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects
$ ls
Djurparken/  pythonProject/

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects
$ cd Djurparken/

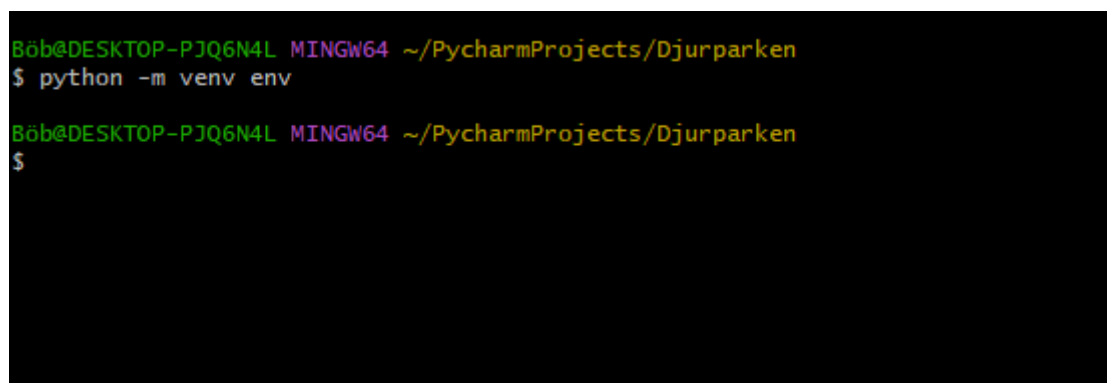
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken
$ ls
assets/  docs/  src/  tests/

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken
$ |
```

Här ska vi då installera en Python-miljö.

Skriv in följande:

python -m venv env



```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken
$ python -m venv env

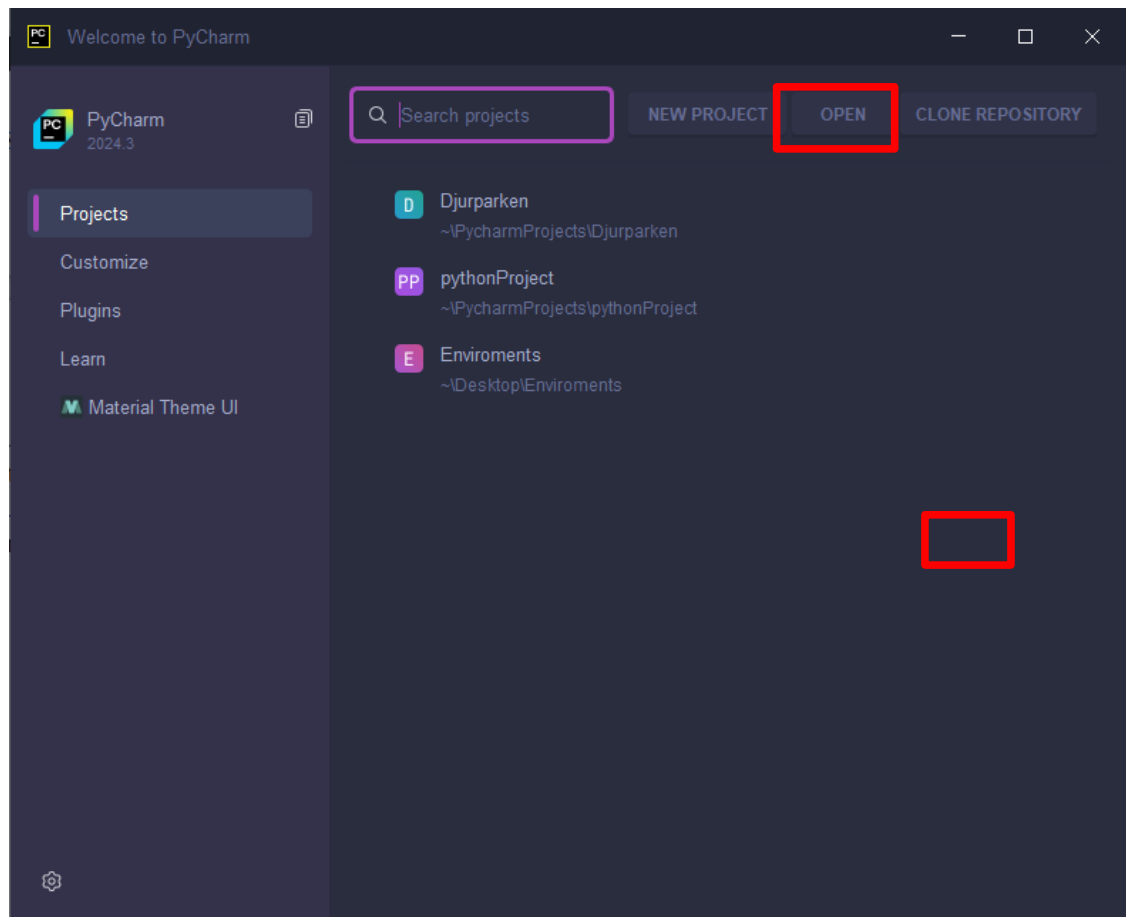
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken
$
```

Kan skilja sig om du behöver skriva python eller python3.

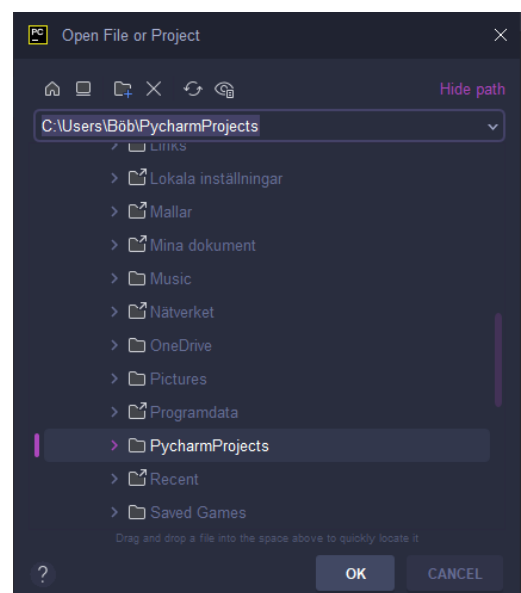
PyCharm

Nu ska vi då starta Pycharm och öppna detta projekt. Du kan ju skapa mappar, virtuell miljö i Pycharm också. Så du måste inte använda terminalen. Jag kommer inte att visa hur du gör detta med Pycharm.

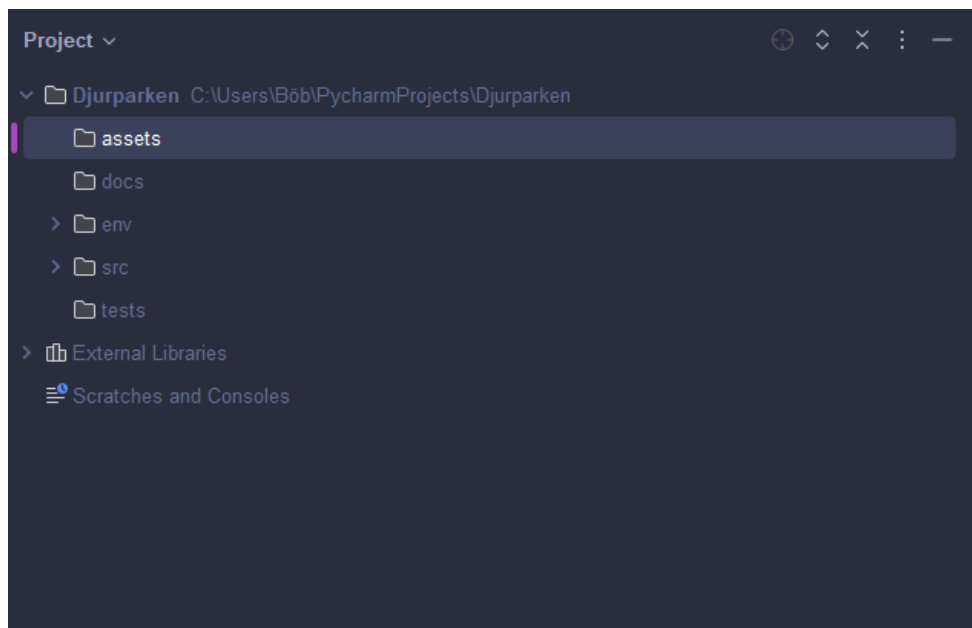
Öppna Pycharm. Här tar du Open.



Letar sedan upp där du har skapat ditt projekt.
Och klickar på OK.

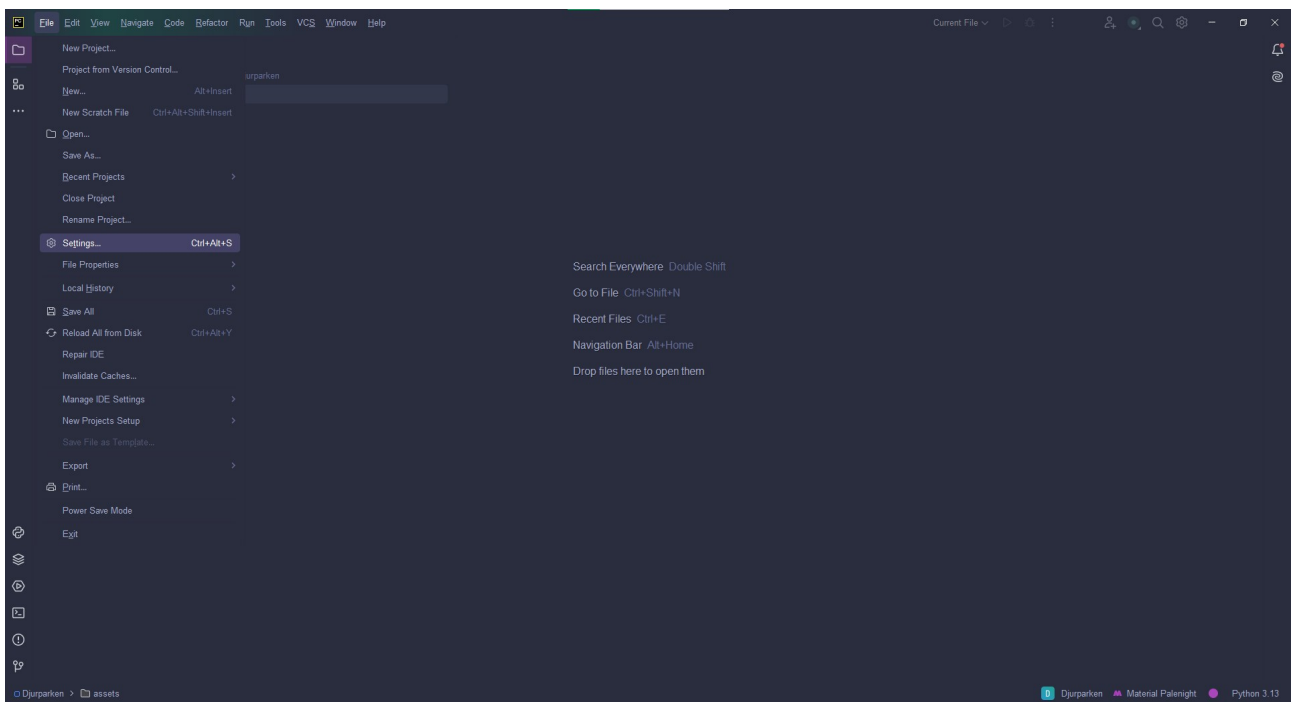


Här ser vi nu då mapparna vi har skapat och vår virtuella-miljö.



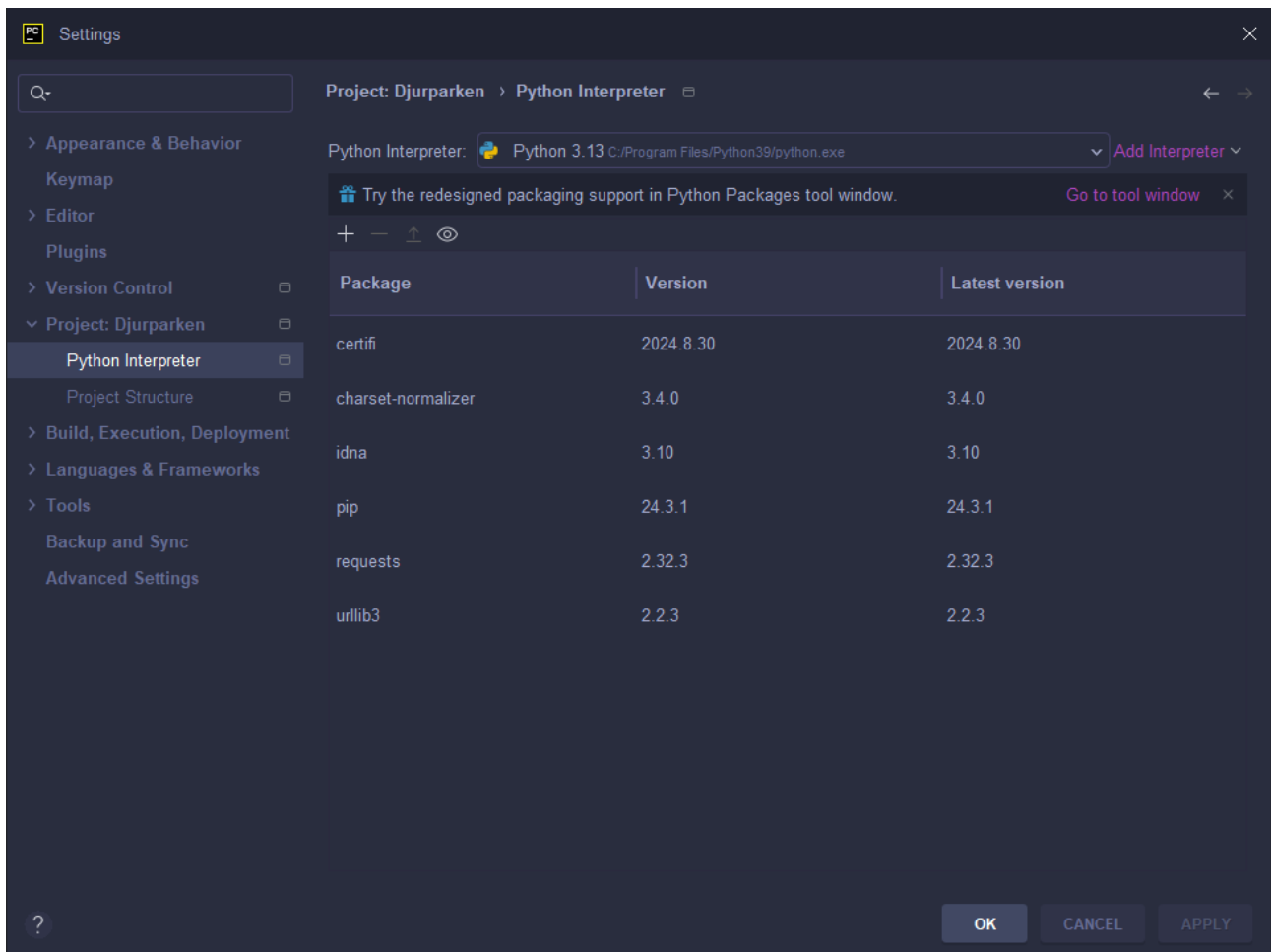
Om vi sedan klickar på ctrl-alt-S

Får vi upp settings. Annars ser du på bilden nedan vart du hittar settings.

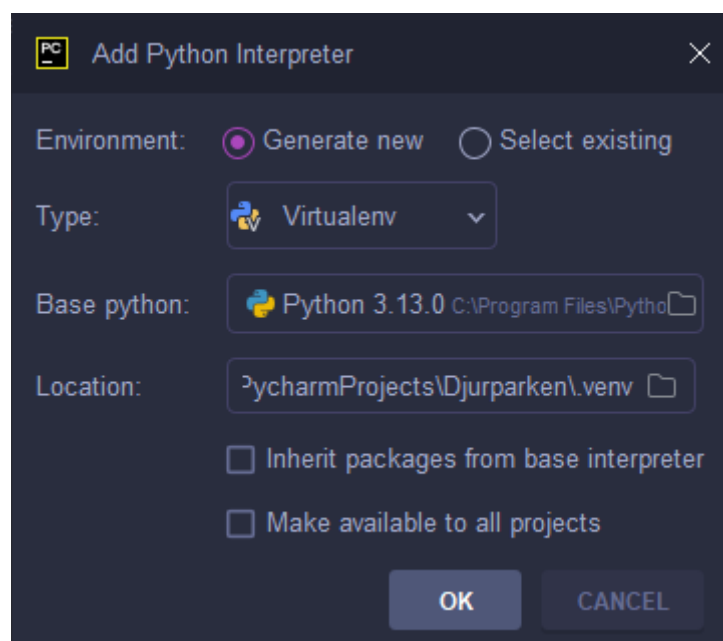


Sedan i vänster spalten så finns det Project: Djurparken

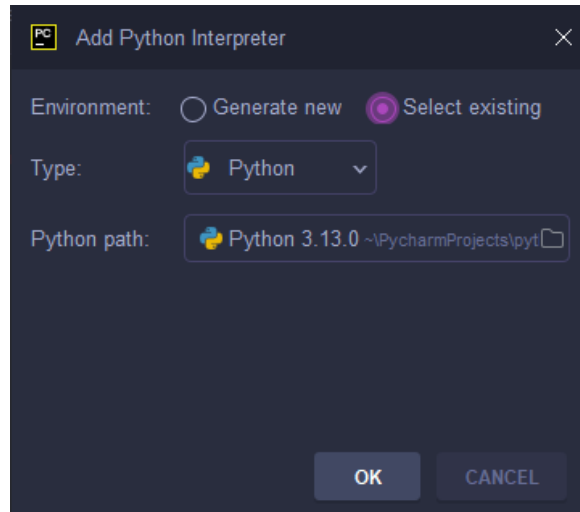
Klicka in där och gå till Python Interpreter



Klicka på Add interpreter uppe till höger. Se figur ovan.

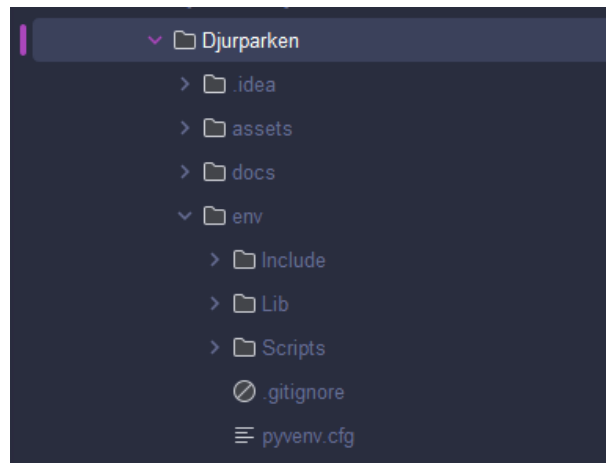


Sedan ta Select existing.

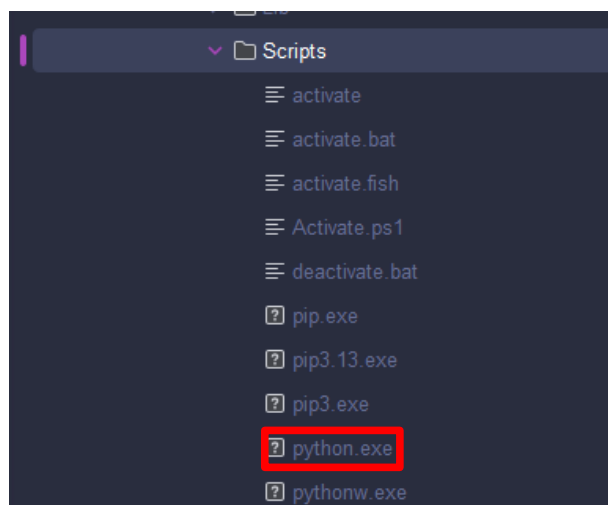


Välj sedan Python path: och välj env mappen i din root-mapp för projektet.

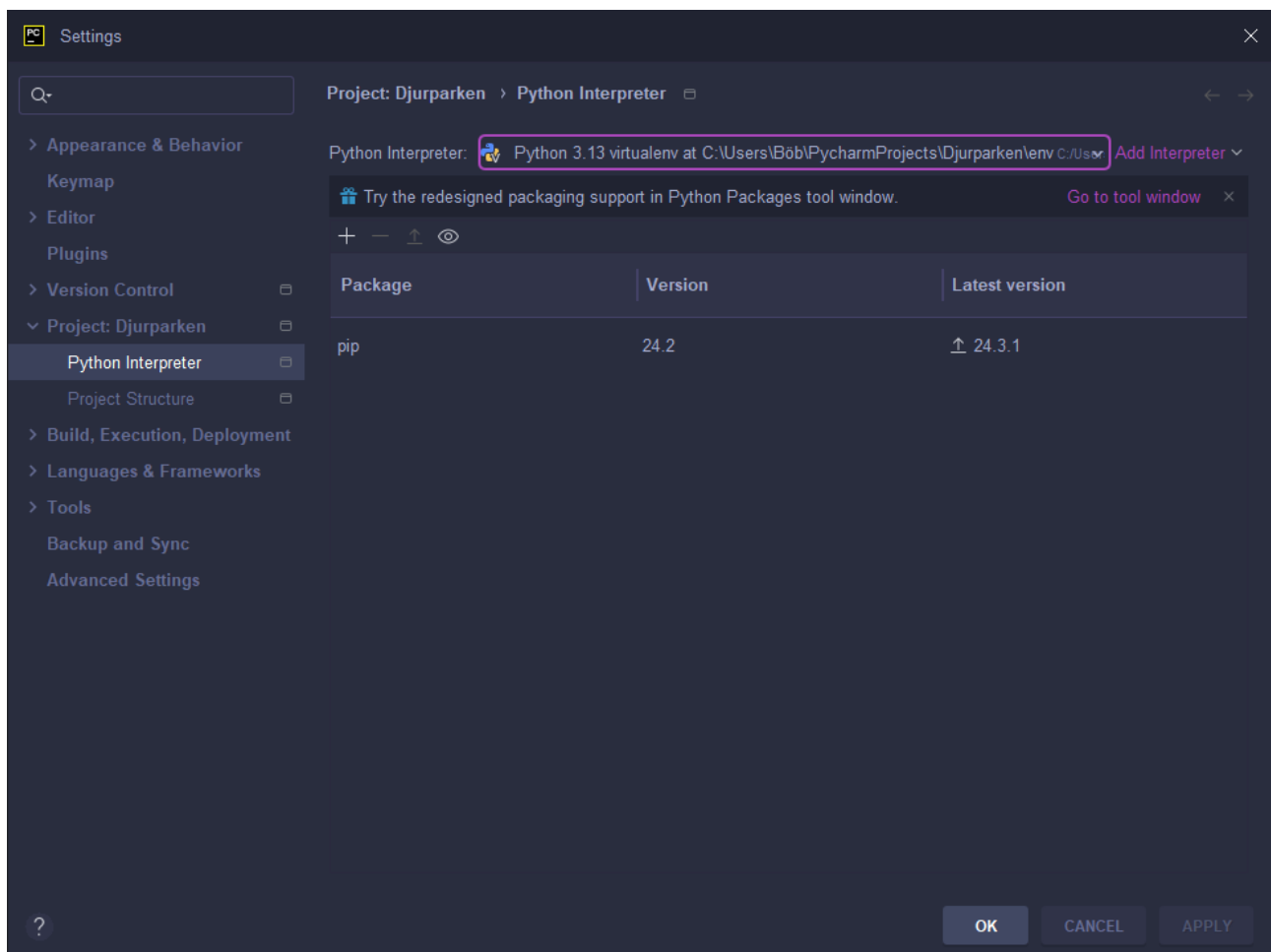
Sen letar du rätt på Scripts.
Här i hittar du execute filen för
python. Som heter python.exe



Där efter klickar du på OK
och efter detta så är projektet
inställt
med miljön du skapade.



För att dubbelkolla detta så kan du gå tillbaka till inställningar och fliken Python Interpreter



Där ser du vilken som är vald. Detta ska nu vara din lokala python interpreter.

Git

Vi ska nu initiera Git i vårt projekt. För att ha versionshantering om vi skulle koda och behöver kanske gå tillbaka till föregående kod innan vi starta. Även då för att kunna lägga upp detta på Github.

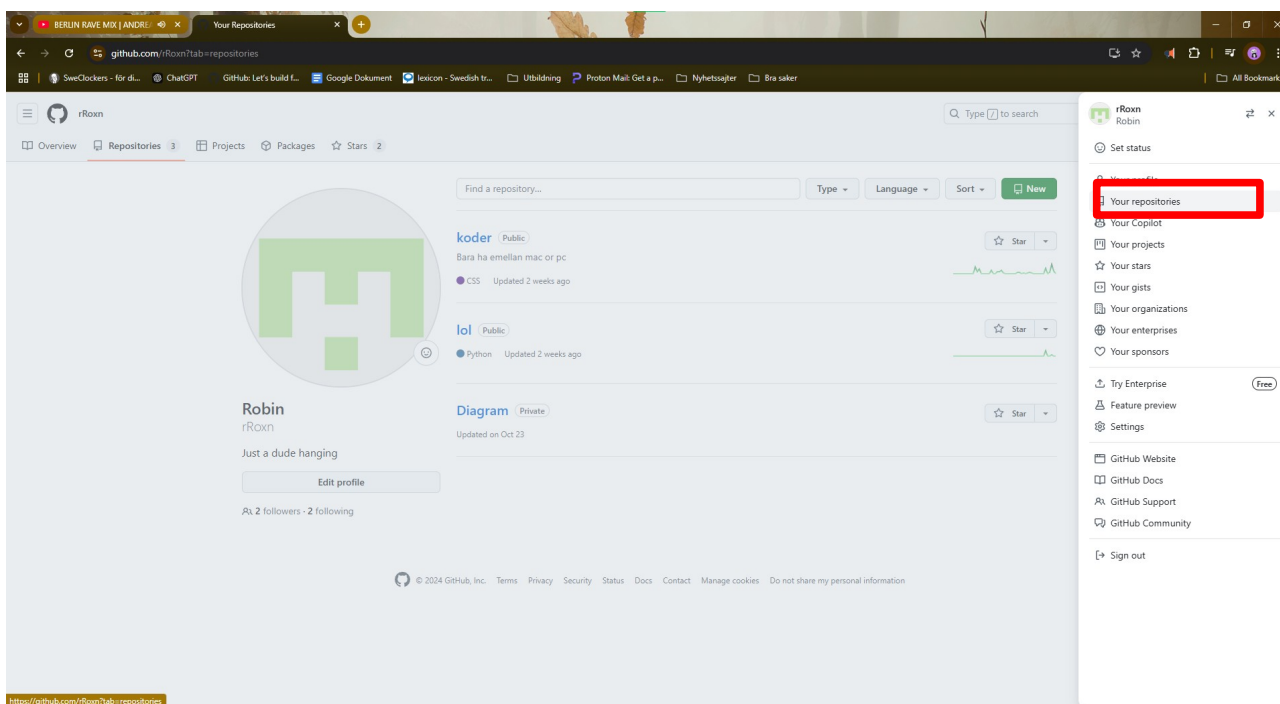
Vi börjar med att skapa en repository på Github.com. Du kan göra detta även med CLI om du laddar ner: <https://cli.github.com/>

Då kan du köra kommandon i din terminal för att skapa repos.

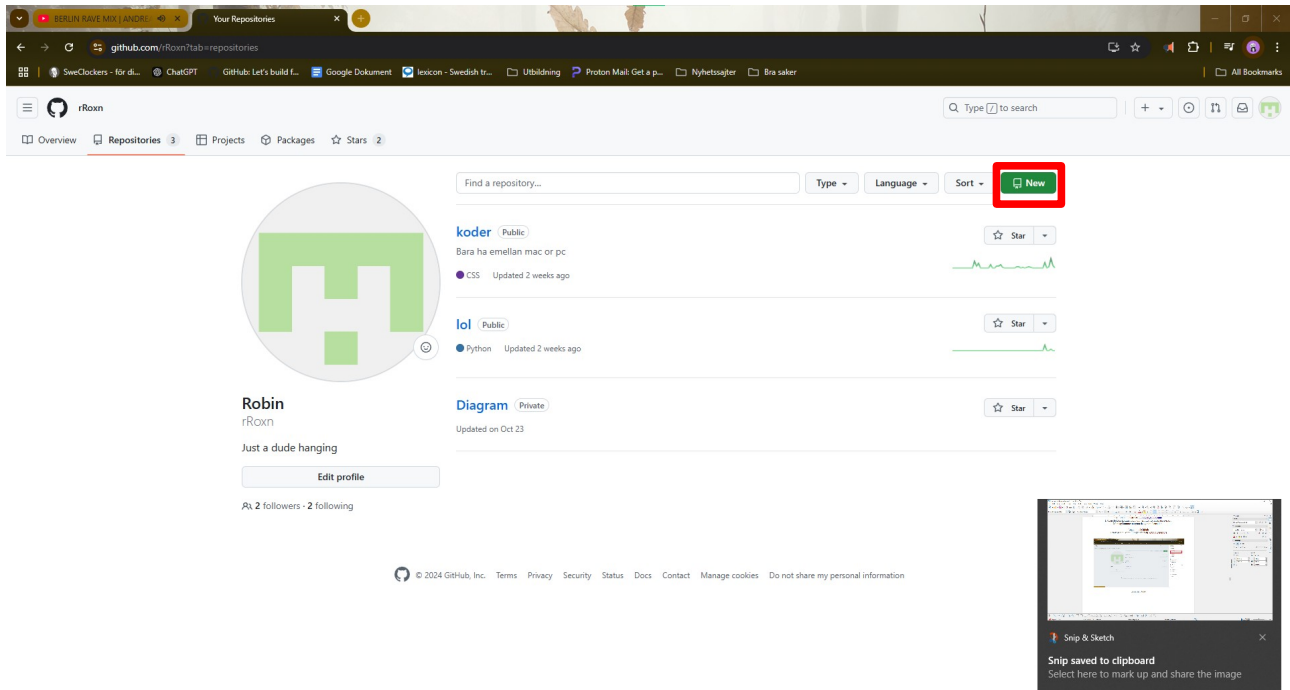
Men jag kommer att göra det genom hemsidan.

Logga in på Github

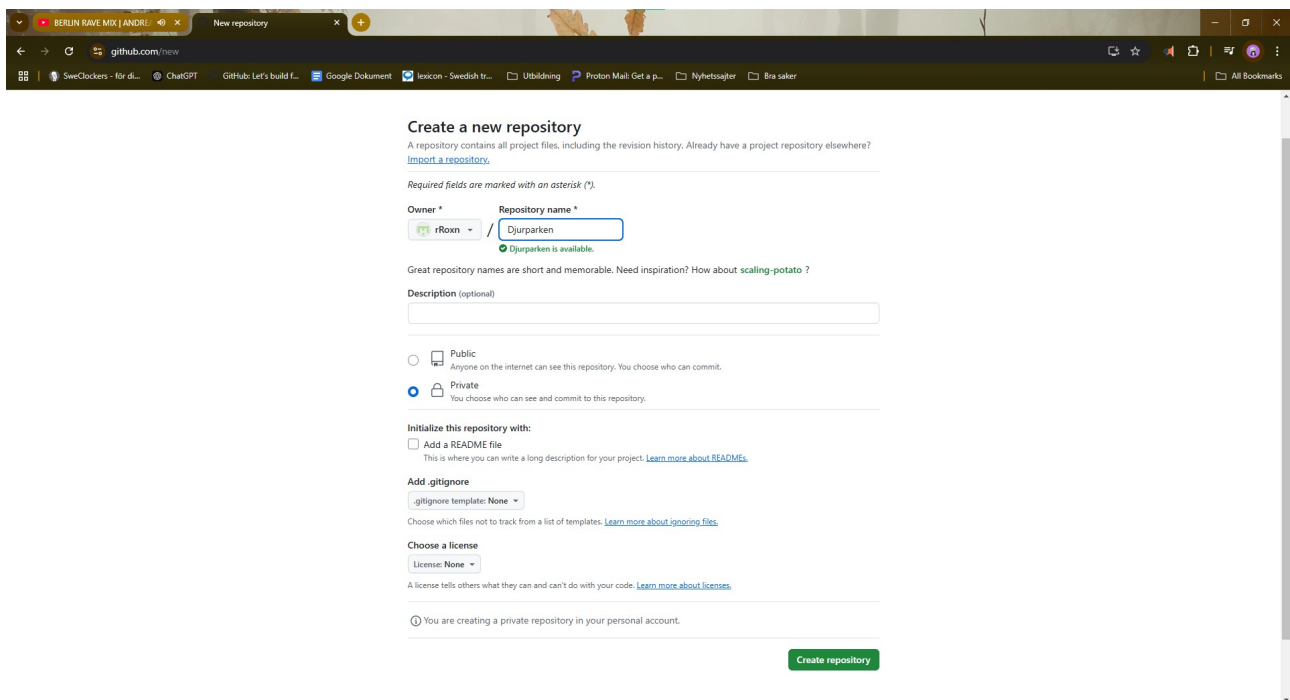
Tryck på din ikon till höger och välj Your repositories



Ta sedan New.

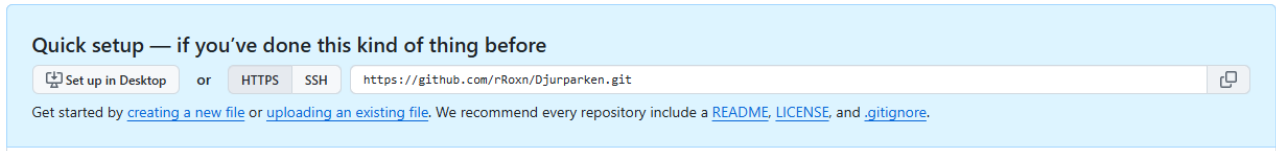


Fyll sedan i Namn för respo, välj sedan private.



Sen klicka Create repository

Sen kopiera HTTPS länken.



Sen går vi till CLI och skriver in:
git remote add origin https://github.com/rRoxn/Djurparken.git

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)
$ git remote add origin https://github.com/rRoxn/Djurparken.git

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)
$
```

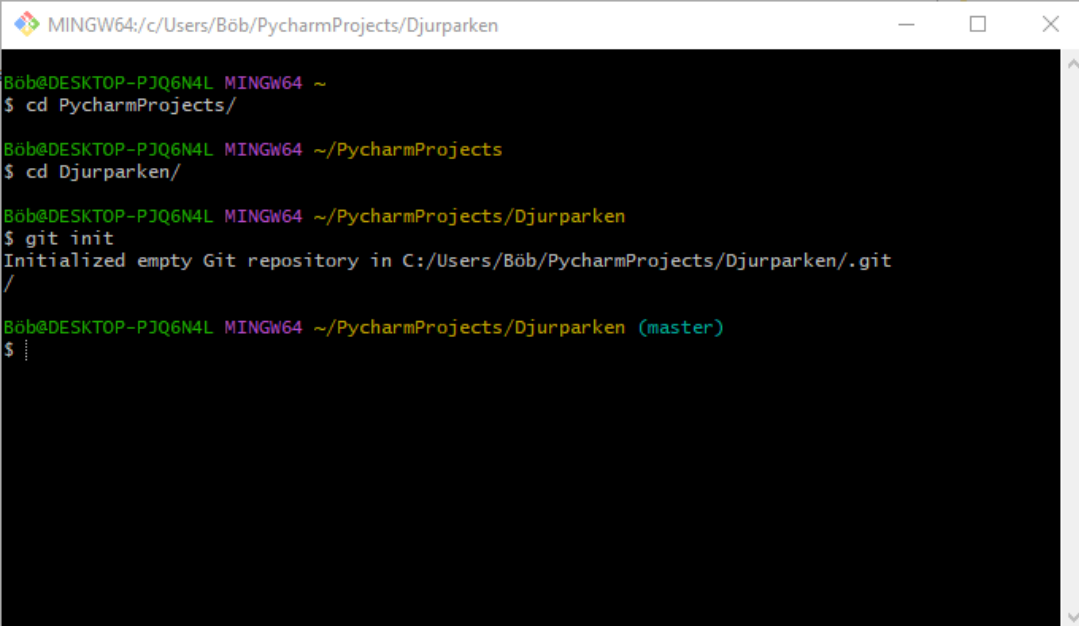
Dubbelkolla sedan så att origin är rätt med:
git remote -v

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)
$ git remote -v
origin https://github.com/rRoxn/Djurparken.git (fetch)
origin https://github.com/rRoxn/Djurparken.git (push)

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)
$
```

Nu ska vi då göra mappen "git" som jag redan har gjort (ni ser att det står (master))
men kommer visa det nu.

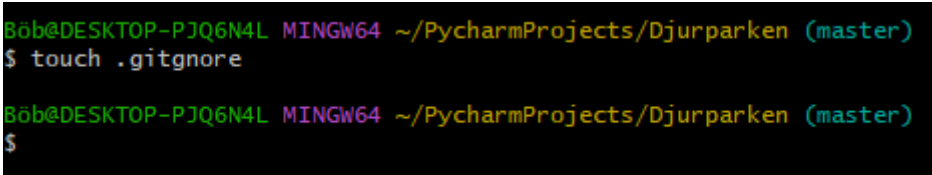
Göra mappen "Git" med kommandot:
git init

A terminal window titled 'MINGW64:/c/Users/Böb/PycharmProjects/Djurparken' with standard window controls. The terminal shows the following commands and output:

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~  
$ cd PycharmProjects/  
  
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects  
$ cd Djurparken/  
  
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken  
$ git init  
Initialized empty Git repository in C:/Users/Böb/PycharmProjects/Djurparken/.git  
/  
  
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)  
$
```

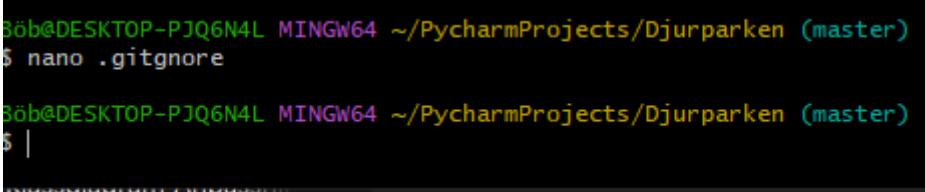
Nu ska vi då skapa en fil som heter .gitignore. Det denna fil gör är att ignorera vissa filer som man lägger till i denna. Så dessa inte laddas upp på din respo.

Vi skriver:
touch .gitignore

A terminal window showing the execution of the 'touch .gitignore' command. The prompt is 'Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)'. The command '\$ touch .gitignore' is entered and executed, resulting in a new prompt '\$' on the next line.

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)  
$ touch .gitignore  
  
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)  
$
```

Skriv sen nano .gitignore för att öppna denna fil i CLI.

A terminal window showing the execution of the 'nano .gitignore' command. The prompt is 'Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)'. The command '\$ nano .gitignore' is entered and executed, resulting in a new prompt '\$ |' on the next line, indicating the nano editor is open.

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)  
$ nano .gitignore  
  
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (master)  
$ |
```


Lägg till följande i denna fil.

```
#Ignorera .idea filen som Pycharm skapar  
.idea/  
  
#Ignorera mitt virtuella miljö  
.env/  
f
```

Nu ska vi då göra vår första commit.

Vi skriver:

git add -all

Sen skriver vi:

git commit -m "Initial commit"

Innan vi gör vår första Push. Ska vi byta namn på vår bransch till main. Master är ett gammalt namn för äldre git versioner.

Git bransch -M main

Jag glömde av att printa detta så har inga bilder på det. Men nu ska vi då göra första pushen till vår respo online.

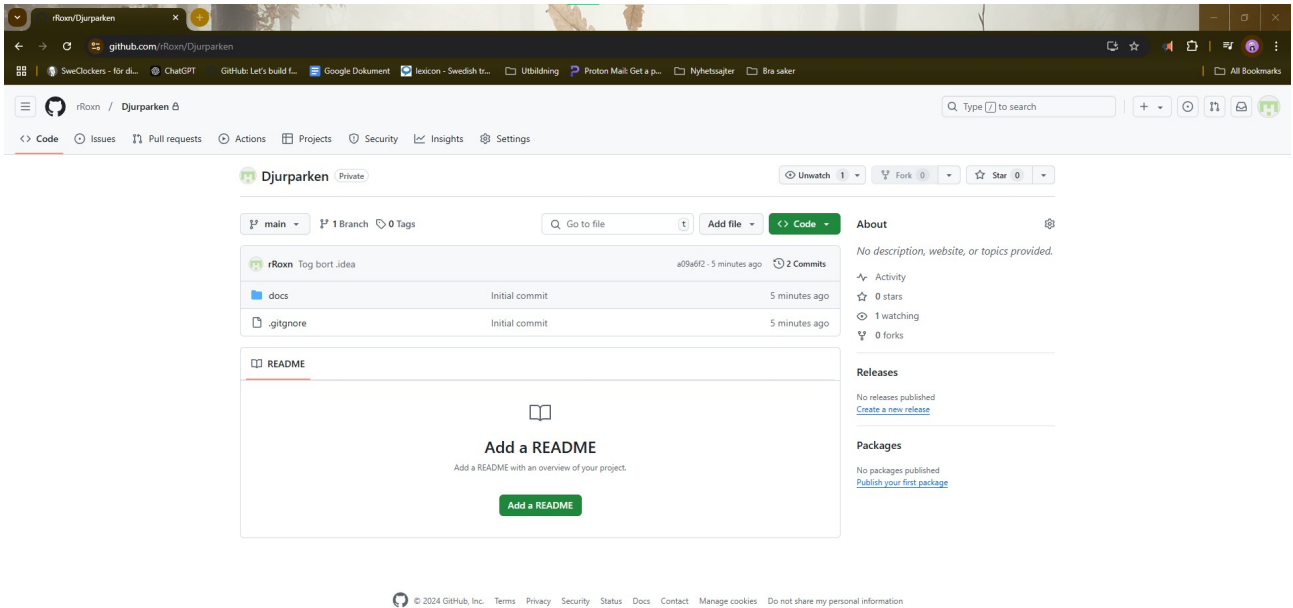
Vi skriver:

git push -u origin main

Förklaring av -u: Den flaggan länkar din lokala main-gren till fjärr-repositoryt, så att framtida git push-kommandon kan göras utan att behöva ange origin och grenen manuellt.

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)  
$ git push -u origin main  
Enumerating objects: 26, done.  
Counting objects: 100% (26/26), done.  
Delta compression using up to 8 threads  
Compressing objects: 100% (24/24), done.  
Writing objects: 100% (26/26), 1.73 MiB | 1.82 MiB/s, done.  
Total 26 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)  
To https://github.com/rRoxn/Djurparken.git  
 * [new branch]      main -> main  
branch 'main' set up to track 'origin/main'.  
  
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)  
$ |
```

Nu går vi in på Github och kollar vår repo.



<https://github.com/rRoxxn/Djurparken>

Nu ser vi att det finns filer här. Då vi inte har skapat filstrukturen ännu. Och jag har endast filer i min docs mapp. Så läggs endast detta upp.

Jag kommer inte att visa mer av just Git. Ville bara ta med hur jag initierade detta. Kanske inte behövdes men tänkte det kunde vara kul att ha med i min dokumentation.

Dokumentation och Requiriements

Implementering och Kod beskrivning

Introduktion

Detta avsnitt beskriver hur programmet Djurparken implementerades, inklusive klasser, metoder och hur programmet uppfyller kravspecifikationen. Fokus ligger på att visa hur de viktigaste funktionerna är designade och fungerar i praktiken. Kommer att visa hur jag skapade filerna. Även kommer all kod finnas med i min Github. Då det blir mycket text av att visa exakt all kod jag skrivit.

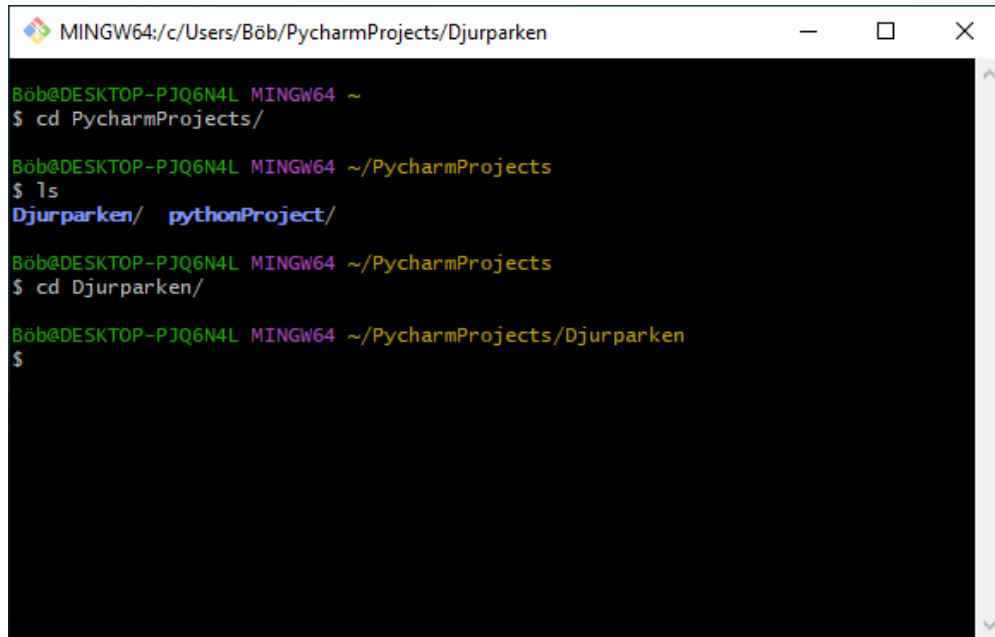
Jag använder även engelska namn på variabler, funktioner etc. Tycker det blir lättare med utvecklingen och kodningen.

Filstruktur

Nu ska vi då skapa filstrukturen för projektet för att underlätta utvecklingen.

Du kan göra detta med hjälp av PyCharm eller terminalen. Jag skapar dessa genom terminalen.

Navigera till din projekt mapp.



```
MINGW64; c:/Users/Böb/PycharmProjects/Djurparken
Böb@DESKTOP-PJQ6N4L MINGW64 ~
$ cd PycharmProjects/

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects
$ ls
Djurparken/  pythonProject/

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects
$ cd Djurparken/

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken
$
```

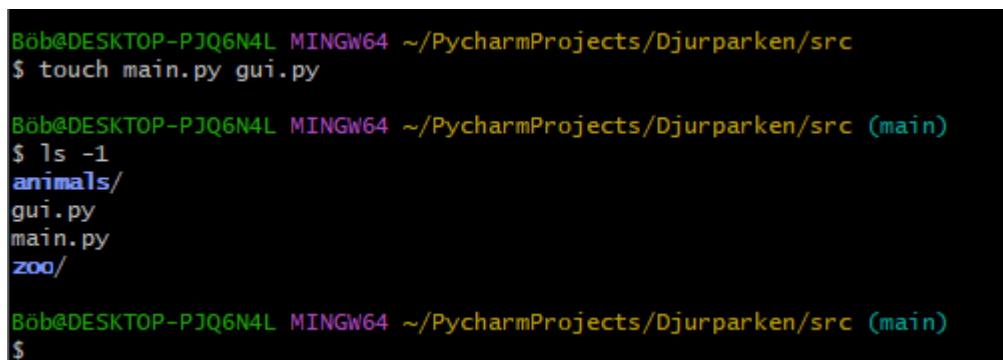
Skapa filer med hjälp utav kommandot touch.

Navigera till src/

Här ska de skapas två filer. Som är main.py, gui.py

Så skriv:

touch main.py gui.py



```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/src
$ touch main.py gui.py

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/src (main)
$ ls -l
animals/
gui.py
main.py
zoo/

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/src (main)
$
```

Navigera in i zoo.

cd zoo

Skriv:

touch __init__.py zoo.py visitors.py

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/src/zoo (main)
$ touch __init__.py zoo.py visitors.py

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/src/zoo (main)
$ ls
__init__.py  visitors.py  zoo.py

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/src/zoo (main)
$
```

Navigera sedan tillbaka och navigera till animals

cd ..

cd animals

touch __init__.py animal.py lion.py lioncub.py giraffe.py elephant.py

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/src/animals (main)
$ touch __init__.py animal.py lion.py lioncub.py giraffe.py elephant.py

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/src/animals (main)
$ ls -l
__init__.py
animal.py
elephant.py
giraffe.py
lion.py
lioncub.py
```

Navigera nu tillbaka till root-mappen. Navigera in på tests.

Cd .. 2 gånger

cd tests

touch test_zoo.py

```

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)
$ cd tests/

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/tests (main)
$ touch test_zoo.py test_animals.py test_visitors.py test_gui.py

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/tests (main)
$ ls -l
test_animals.py
test_gui.py
test_visitors.py
test_zoo.py

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken/tests (main)

```

Nu är då filstrukturen färdig med alla våra filer skapade för projektet. Nu kan vi då börja koda.

Pytest

Jag kommer att använda pytest för mitt projekt. Som är en modul för python för att köra tester på olika funktioner och det gör de lättare med felsökningen. Du slipper skriva dessa i källkoden.

Vi installerar detta genom följande:

pip install pytest

```

Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)
$ pip install pytest
Collecting pytest
  Downloading pytest-8.3.4-py3-none-any.whl.metadata (7.5 kB)
Collecting colorama (from pytest)
  Downloading colorama-0.4.6-py2.py3-none-any.whl.metadata (17 kB)
Collecting iniconfig (from pytest)
  Downloading iniconfig-2.0.0-py3-none-any.whl.metadata (2.6 kB)
Collecting packaging (from pytest)
  Downloading packaging-24.2-py3-none-any.whl.metadata (3.2 kB)
Collecting pluggy<2,>=1.5 (from pytest)
  Downloading pluggy-1.5.0-py3-none-any.whl.metadata (4.8 kB)
Downloading pytest-8.3.4-py3-none-any.whl (343 kB)
Downloading pluggy-1.5.0-py3-none-any.whl (20 kB)
Downloading colorama-0.4.6-py2.py3-none-any.whl (25 kB)
Downloading iniconfig-2.0.0-py3-none-any.whl (5.9 kB)
Downloading packaging-24.2-py3-none-any.whl (65 kB)
Installing collected packages: pluggy, packaging, iniconfig, colorama, pytest
Successfully installed colorama-0.4.6 iniconfig-2.0.0 packaging-24.2 pluggy-1.5.0 p
3.4

```

Jag kommer även då att skapa en requirements.txt. Där denna fil är för att veta vad för moduler och vad du behöver ha för att kunna köra detta program. Detta är för min användning då jag byter dator ganska ofta.

Vi initierar detta genom att skriva:

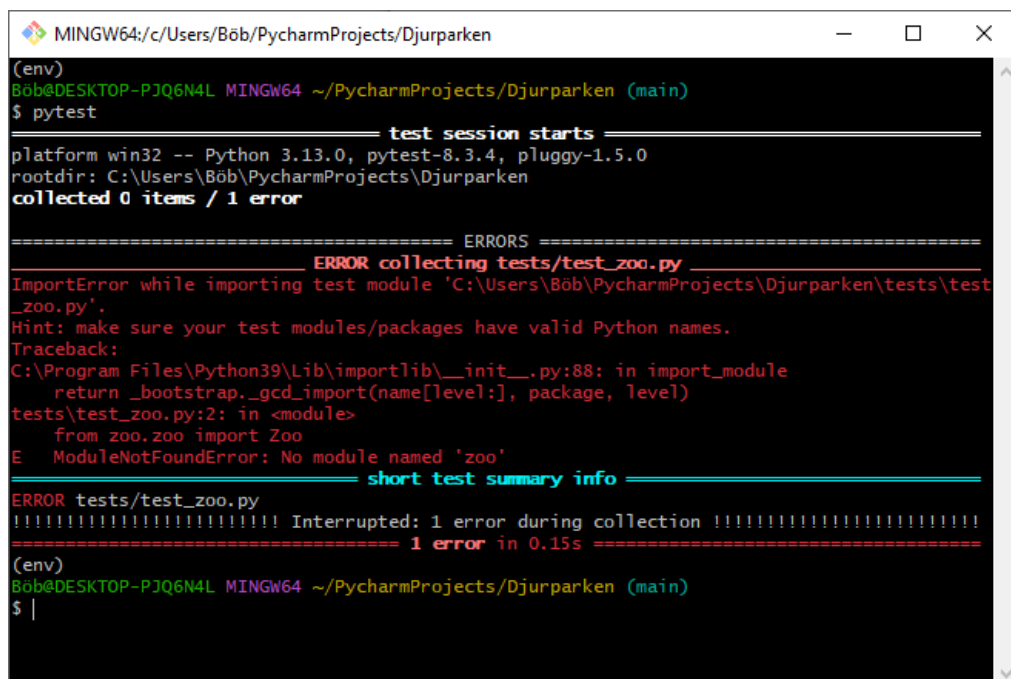
```
echo pytest > requirements.txt
```

```
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)
$ echo pytest > requirements.txt
(env)
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)
$ ls
assets/ docs/ env/ requirements.txt src/ tests/
(env)
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)
$ |
```

Detta skapar då en ny fil och skriver in pytest i denna. Om jag nu installerar flera python paket så kommer jag att skriva in dessa här.

Hur man kör pytest är genom att skriva pytest i terminalen.

Här kan man då felsöka och ser vad som blir fel vid körningar etc.



```
MINGW64;c:/Users/Böb/PycharmProjects/Djurparken
(env)
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)
$ pytest
===== test session starts =====
platform win32 -- Python 3.13.0, pytest-8.3.4, pluggy-1.5.0
rootdir: C:\Users\Böb\PycharmProjects\Djurparken
collected 0 items / 1 error

===== ERRORS =====
ERROR collecting tests/test_zoo.py
ImportError while importing test module 'C:\Users\Böb\PycharmProjects\Djurparken\tests\test_zoo.py'.
Hint: make sure your test modules/packages have valid Python names.
Traceback:
C:\Program Files\Python39\Lib\importlib\__init__.py:88: in import_module
    return _bootstrap._gcd_import(name[level:], package, level)
tests\test_zoo.py:2: in <module>
    from zoo.zoo import Zoo
E   ModuleNotFoundError: No module named 'zoo'

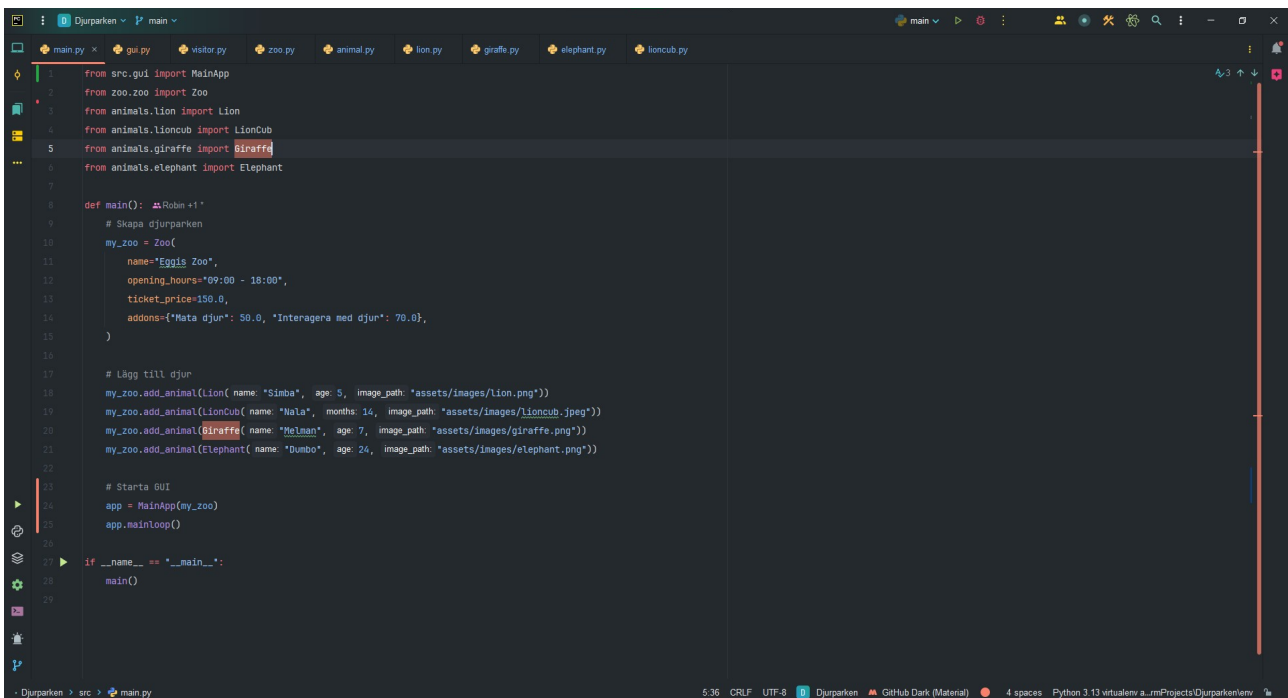
===== short test summary info =====
ERROR tests/test_zoo.py
!!!!!!!!!!!!!!!!!!!! Interrupted: 1 error during collection !!!!!!!!!!!!!!!!!!!!!
===== 1 error in 0.15s =====
(env)
Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)
$ |
```


Kodbeskrivning: Av de olika filerna i programmet

I det här avsnittet beskriver jag hur programmet är uppbyggt, med fokus på klasser, metoder och deras ansvar. Jag har använt objektorienterad programmering (OOP) för att strukturera koden.

Jag kommer att gå igenom hur jag började implementera funktionerna och hur de olika filerna hänger ihop. Genom att förklara varje kodfil och de funktioner som ingår hoppas jag ge en översiktlig bild av hur programmet är organiserat. I slutet av avsnittet kommer jag också kort att beskriva hur jag arbetade med att utveckla dessa filer och sättet jag gick tillväga.

Main.py(huvudfilen)



```
1 from src.gui import MainApp
2 from zoo.zoo import Zoo
3 from animals.lion import Lion
4 from animals.lioncub import LionCub
5 from animals.giraffe import Giraffe
6 from animals.elephant import Elephant
7
8 def main():
9     # Skapa djurparken
10     my_zoo = Zoo(
11         name="Eggis Zoo",
12         opening_hours="09:00 - 18:00",
13         ticket_price=150.0,
14         addons={"Mata djur": 50.0, "Interagera med djur": 70.0},
15     )
16
17     # Lägg till djur
18     my_zoo.add_animal(Lion(name="Simba", age=5, image_path="assets/images/lion.png"))
19     my_zoo.add_animal(LionCub(name="Hala", months=14, image_path="assets/images/lioncub.jpeg"))
20     my_zoo.add_animal(Giraffe(name="MeLian", age=7, image_path="assets/images/giraffe.png"))
21     my_zoo.add_animal(Elephant(name="Dumbo", age=24, image_path="assets/images/elephant.png"))
22
23     # Starta GUI
24     app = MainApp(my_zoo)
25     app.mainloop()
26
27 if __name__ == "__main__":
28     main()
```

Denna fil är huvudfilen som kör programmet. Här anropas användargränssnittet (GUI) som hanterar visningen av djurparken. Filen innehåller följande sektioner:

1. **Imports:** Alla nödvändiga moduler och klasser importeras från externa filer och bibliotek, inklusive de som definierar djurklasser och GUI-komponenter.
2. **Funktionen main:** Denna funktion skapar instansen av djurparken med definierade parametrar, såsom namn, öppettider, biljettpris och tillval.
3. **Hårdkodade djur:** Djur läggs till i djurparken genom att anropa `add_animal` metoden och skicka med specifika attribut, inklusive namn, ålder och bildväg.
4. **Start av GUI:** Slutligen startas GUI:t genom att skapa en instans av `MainApp` med djurparken som parameter, och programmet körs i en oändlig loop via `app.mainloop()`.

Denna fil fungerade också som meny för programmet under utvecklingen när TUI användes istället för GUI.

```
def __init__(self, name: str, opening_hours: str, ticket_price: float, addons: dict):  🧑 Robin
    """Initierar djurparken."""
    self.name = name
    self.opening_hours = opening_hours
    self.ticket_price = ticket_price
    self.addons = addons
    self.animals = [] # Lista över alla djur
    self.visitors = [] # Lista över besökare, max 3
```

Konstruktorn för Djurpark-klassen

I denna del av koden definieras konstruktorn (`__init__`) för att skapa en instans av djurparken. När programmet startas, anropas denna konstruktor med de angivna argumenten: namn, öppettider, biljettpris och tillval. De olika variablerna som sätts är:

1. **Namn:** Namnet på djurparken.
2. **Öppettider:** Djurparkens öppettider definieras som en sträng.
3. **Biljettpris:** Det pris som besökare ska betala för inträde.
4. **Tillval:** Ett ordbok (dictionary) som innehåller tillgängliga tillval för besökare, som t.ex. extra aktiviteter.

Det finns också två listor som används för att hålla reda på:

- **animals:** En lista som lagrar alla djur i djurparken.
- **visitors:** En lista som håller reda på besökare, med ett maxantal på tre besökare åt gången.

Vissa av variablerna har satts som skyddade (protected) eller privata (private) attribut för att säkerställa att de inte ändras direkt från externa källor, vilket ger en bättre kapsling och kontroll över objektets tillstånd.

I bilden nedan ser vi ett par av funktionerna i denna klass, det kommer beskrivningar nedan för dessa.

```
def add_animal(self, animal): 7 usages (3 dynamic) 🐞Robin
    """Lägger till ett djur i djurparken."""
    self.animals.append(animal)

def list_animals(self, detailed=False): 🐞Robin
    """Returnerar en lista med förenklad eller detaljerad information om djuren."""
    if detailed:
        return [str(animal) for animal in self.animals] # Full info med __str__
    else:
        return self.animals # Returnera djurobjekt

def show_ticket_price(self) -> str: 1 usage (1 dynamic) 🐞Robin
    """Returnerar biljettpris."""
    return f"Biljettpris: {self.ticket_price} SEK."

def show_opening_hours(self) -> str: 1 usage (1 dynamic) 🐞Robin
    """Returnerar öppettider för djurparken."""
    return f"Öppettider: {self.opening_hours}."

def show_addon_prices(self) -> str: 🐞Robin
    """Visar tillgängliga tillval och deras priser."""
    return "\n".join(f"{addon}: {price} SEK" for addon, price in self.addons.items())

def feed_animal(self, name: str, food: str) -> str: 1 usage (1 dynamic) 🐞Robin
    """Matar ett djur i djurparken."""
    for animal in self.animals:
        if animal.name.lower() == name.lower():
            return animal.eat(food)
    return f"Inget djur med namnet '{name}' hittades."
```

Här är en kort fyra funktionerna i koden:

1. **add_animal(self, animal):**

- Denna funktion lägger till ett djur i djurparken genom att använda `append()` för att lägga djuret i listan `self.animals`. Det gör att djuret kan hanteras inom programmet.

2. **list_animals(self, detailed=False):**

- Denna funktion returnerar en lista över alla djur i djurparken. Om argumentet `detailed` sätts till `True`, returneras en detaljerad lista med all information om djuren. Om det är `False` returneras endast djurobjekten i en förenklad lista.

3. **show_ticket_price(self):**

- Funktionen returnerar biljettpriset för djurparken i ett format som visar priset i SEK.

4. **show_opening_hours(self):**

- Denna funktion returnerar öppettiderna för djurparken i en läsbar sträng som beskriver när djurparken är öppen.

5. **feed_animal(self, name, food):**

- Denna funktion tillåter en besökare att mata ett djur. Funktionen söker efter ett djur med namnet som ges som argument. Om djuret hittas, anropas djurets eat metod och djuret matas. Om inget djur med det namnet finns, returneras ett felmeddelande.

Dessa funktioner är grundläggande för att hantera djur och information om djurparken i ditt program.

Animal.py

```
from abc import ABC, abstractmethod

class Animal(ABC):  # Robin +1 *
    def __init__(self, name: str, age: int, favorite_food: str, image_path: str):  # Robin +1 *
        """Initierar ett djur."""
        |
        self.name = name
        self.age = age
        self.favorite_food = favorite_food
        self.hungry = True
        self.image_path = image_path

    @abstractmethod  # 5 usages (5 dynamic)  # Robin +1
    def interact(self) -> str:
        """Interagerar med djuret. Måste implementeras av subklasser."""
        pass

    @abstractmethod  # 5 usages (5 dynamic)  # rRoxn +1
    def eat(self, food: str) -> str:
        """Funktion för att mata djuren. Måste implementeras av subklasser."""
        pass
```

Den här klassen fungerar som basklass för alla djur i djurparken. Den är definierad som en abstrakt klass, vilket innebär att den inte kan instansieras direkt. Alla subklasser (t.ex. Lion, Elephant) måste ärva från denna klass och implementera de abstrakta metoderna.

- **Konstruktorn:** I konstruktorfunktionen (`__init__`) definieras djurets grundläggande egenskaper som namn, ålder, favoritmat och bildväg. Alla djur instansieras med dessa attribut.
- **Abstrakta metoder:** Två abstrakta metoder är definierade för att säkerställa att alla subklasser implementerar specifika funktioner:
 - **`interact(self)`:** En metod som definierar interaktionen med djuret. Denna metod måste implementeras i varje subklass för att specificera hur djuret kan interagera med besökare.
 - **`eat(self, food: str)`:** En metod för att mata djuret. Subklasserna måste implementera denna metod för att definiera hur djuret ska äta.

Användningen av abstrakta metoder säkerställer att alla djurklasser har en gemensam struktur, men tillåter olika implementeringar för varje djurspecifik funktionalitet.

Animal sub-klasser

```
class Lion(Animal): 3 usages 🧑Robin +1 *
    def __init__(self, name: str, age: int, image_path: str): 🧑Robin +1
        """Initierar ett lejon."""
        super().__init__(name, age, "kött", image_path)

    def interact(self) -> str: 5 usages (5 dynamic) 🧑Robin +1
        """Interaktion för lejon."""
        self.hungry = True # Interaktion gör lejonet hungrigt
        return f"{self.name} tittar stolt på dig."

    def eat(self, food: str) -> str: 5 usages (5 dynamic) 🧑rRoxn +1 *
        """Lejonets specifika sätt att äta."""
        if not self.is_hungry(): # Använd basklassens metod för att kontrollera hunger
            return f"{self.name} är inte hungrig."
        if self.validate_food(food): # Kontrollera om det är favoritmat
            self.toggle_hunger() # Växla hungerstatus med basklassens metod
            return f"{self.name} sliter i sig {food} med en kraftig riv!"
        return f"{self.name} rynkar på nosen och vägrar äta {food}."

    def get_species(self) -> str: 1 usage (1 dynamic) 🧑Robin
        """Returnerar artens namn."""
        return "Lejon"
```

I denna del av programmet definieras specifika djurklasser som ärver från basklassen Animal. Varje djurklass implementerar de abstrakta metoderna interact och eat för att definiera djurens specifika beteenden.

1. Lion klass:

- **Konstruktorn (__init__):** Initierar ett lejon med specifika attribut som namn, ålder, favoritmat (kött) och bildväg.
- **interact(self):** När man interagerar med lejonet, sätts dets hungerstatus till True och lejonet tittar stolt på besökaren.
- **eat(self, food: str):** Lejonet har en särskild metod för att äta. Först kontrolleras om det är hungrigt. Om det inte är hungrigt, avböjer lejonet maten. Om det är hungrigt, kontrolleras om maten är dess favorit. Om det inte är det, rynkar lejonet på nosen och vägrar äta.

2. LionCub klass:

- Denna klass ärver från Lion, vilket gör att den automatiskt ärver vissa metoder från lejonklassen. Skillnaden är att det är ett lejonunge och kan ha specifika variationer i beteende som kan definieras i subklassen.

3. Giraffe och Elephant klasser:

- Dessa klasser fungerar på samma sätt som Lion-klassen. Varje djur har sina egna implementationer av interact och eat-metoderna som anpassas för deras specifika beteenden. T.ex. giraffen kan sträcka sin hals för att få tag på mat och elefanten kan ha en annan metod för att äta eller interagera.

Generell Struktur:

- Alla djurklasser ärvda från Animal säkerställer att de har gemensamma attribut som name, age, favorite_food och image_path.
- Genom att använda abstrakta metoder säkerställs det att alla djurklasser implementerar sina egna versioner av interact och eat, vilket ger dem unika beteenden samtidigt som de följer en gemensam struktur.


```
class Visitor: 2 usages  🧑 Robin
    def __init__(self, name: str, budget: float):  🧑 Robin
        """Initierar en besökare."""
        self.name = name
        self.budget = budget
        self.cart = [] # Lista över valda tillval

    def add_to_cart(self, item: str, price: float) -> bool: 3 usages (3 dynamic)  🧑 Robin
        """Lägger till ett tillval i kundvagnen om budgeten räcker."""
        if self.budget >= price:
            self.cart.append((item, price))
            self.budget -= price
            return True # Returnera True om tillvalet lades till
        return False # Returnera False om budgeten inte räckte

    def summarize_cart(self) -> str:  🧑 Robin
        """Visar kundvagnen och totalkostnaden."""
        if not self.cart:
            return "Kundvagnen är tom."
        total_cost = sum(item[1] for item in self.cart)
        items = "\n".join(f"{item[0]}: {item[1]} SEK" for item in self.cart)
        return f"Din kundvagn:\n{items}\nTotalpris: {total_cost} SEK"
```

Denna klass representerar en besökare i djurparken. Varje besökare har ett namn, en budget och en kundvagn där tillval (som biljetter eller aktiviteter) kan läggas till. Besökaren kan köpa tillval så länge deras budget räcker.

1. Konstruktorn (`__init__`):

- **Attribut:**

- name: Besökarens namn.
- budget: Besökarens tillgängliga budget för köp.
- cart: En lista som håller reda på de tillval som besökaren har lagt till i kundvagnen.

2. `add_to_cart(self, item: str, price: float) -> bool`:

- Denna funktion lägger till ett tillval (t.ex. aktivitet eller biljett) i kundvagnen om besökarens budget räcker för priset.
- Om budgeten räcker, läggs tillvalet till i kundvagnen och priset dras av från budgeten. Funktionen returnerar True om tillvalet lades till.
- Om budgeten inte räcker, returneras False.

3. **summarize_cart(self) -> str:**

- Denna funktion summerar innehållet i kundvagnen och visar den totala kostnaden.
- Om kundvagnen är tom, returneras ett meddelande om att den är tom.
- Om det finns tillval i kundvagnen, returneras en lista över dessa tillval samt den totala kostnaden.

gui.py

De här är då koden för min GUI för programmet. Denna blev inte snygg för fem öre då jag inte haft tiden att fixa design. Tänkte mer på HCI som är hur vi människor integrerar med datorer. Hur lätt ser du vart man backar eller hur lätt kan man registrera sig. Så tänkte mer på detta än design.

Denna fil har massor av kod. Då jag gjorde en fil för hela gui och inte flera moduler. Så denna är lite rörig har kommentera det mesta i den. Har även använt mig av grid mestadels till att göra designen. Det har gått sådär och är inte jätte nöjd med slutresultatet. Jag kommer att visa hur detta ser ut längre fram i dokumentationen.

```
class MainApp(tk.Tk): 2 usages  🐞 Robin *
    def __init__(self, zoo):  🐞 Robin *
        super().__init__()
        self.zoo = zoo
        self.title("Djurparken")
        self.geometry("800x600")
        self.frames = {}

        # Container för alla sidor
        container = tk.Frame(self)
        container.grid(row=0, column=0, sticky="nsew") # Ändra från pack till grid

        # Gör att container expanderar
        self.grid_rowconfigure(index=0, weight=1)
        self.grid_columnconfigure(index=0, weight=1)
        container.grid_rowconfigure(index=0, weight=1)
        container.grid_columnconfigure(index=0, weight=1)

        # Öppna i fullskärm
        self.state('zoomed')
```

MainApp-klass: Denna klass är en subclass till tk.Tk och definierar huvudfönstret för applikationen.

1. **Konstruktör (init):** Här initieras fönstret, titel och storlek sätts, och ett zoo-objekt tilldelas till instansvariabeln self.zoo.
2. **Container för sidor:** En Frame-widget används som en behållare för alla sidor i applikationen. Den placeras med hjälp av grid layout.
3. **Expanderbar container:** Genom att använda grid_rowconfigure och grid_columnconfigure görs containern flexibel så att den kan anpassa sig till storleken på fönstret.
4. **Fullskärmsläge:** Applikationen startar i fullskärmsläge.

Denna struktur används för att skapa en flexibel och skalbar GUI för applikationen, vilket gör det möjligt att hantera olika sidor och interaktioner effektivt.

Arbetsprocess och utvecklingsprocess

När jag började arbeta med koden för djurparksapplikationen fokuserade jag på att skapa en tydlig struktur för både programmets logik och användargränssnitt. Mitt mål var att bygga ett objektorienterat program som är lätt att förstå och enkelt att vidareutveckla. För att lyckas med detta delade jag upp arbetet i flera steg:

1. Design och planering

Jag startade med att skapa en övergripande plan och ett UML-diagram för att se hur alla delar skulle hänga ihop. Det hjälpte mig att identifiera vilka klasser jag behövde och hur de skulle struktureras för att fungera tillsammans.

2. Implementering av grunderna

När planen var på plats började jag med de grundläggande klasserna. Jag skapade först en basklass för djuren, *Animal*, och byggde sedan vidare med specifika djurklasser som *Lion*, *Giraffe* och *Elephant*. Varje klass fick funktioner och egenskaper som behövdes för att hantera matning och interaktion med djuren.

3. Utveckling av användargränssnittet

För att göra programmet användarvänligt utvecklade jag ett grafiskt gränssnitt (GUI) med Tkinter. Här lade jag till funktioner som att låta användaren interagera med djuren, köpa biljetter och tillval samt se information om djuren.

4. Testning och justeringar

Under hela utvecklingen testade jag koden steg för steg för att se att allt fungerade som det skulle. Vid behov gjorde jag justeringar för att förbättra både funktionalitet och användarupplevelse.

5. Sammanfattning av processen

Genom att arbeta stegvis och hålla koden enkel och strukturerad lyckades jag skapa en flexibel och användbar applikation. Jag såg till att alla delar – från djurklasserna till gränssnittet – fungerade bra tillsammans.

Utvecklingsprocess

Det första steget i utvecklingsarbetet var att bygga upp de viktigaste delarna av koden för att få en fungerande struktur. Jag började med att skapa de grundläggande klasserna.

Jag började med att skapa en Zoo-klass, denna klass hanterar listan med djuren, och grundläggande logik för djurparken.

När jag började utveckla **Zoo-klassen** var målet att skapa en central plats för att hantera djurparken och dess funktioner. Klassen skulle vara flexibel nog att hantera djur, besökare och andra delar av programmet. Jag arbetade stegvis för att bygga upp klassen:

Jag började med att definiera **attribut** som behövdes för att representera en djurpark(även kolla då mitt uml-diagram som jag skapat):

- name: Djurparkens namn.
- opening_hours: Parkens öppettider.
- ticket_price: Grundpriset för biljetter.
- addons: En dictionary som innehåller tillval och deras priser.
- animals: En lista för att lagra alla djur som finns i djurparken.
- visitors: En lista för att hantera besökare i parken.

```
class Zoo: 3 usages 🐼Robin
    def __init__(self, name: str, opening_hours: str, ticket_price: float, addons: dict): 🐼Robin
        """Initierar djurparken."""
        #Sätter alla dessa variabler till __private för jag vill ej ändra dessa eller råka gör detta.
        self.__name = name
        self.__opening_hours = opening_hours
        self.__ticket_price = ticket_price
        self.__addons = addons
        self.__animals = [] # Lista över alla djur privat
        self.__visitors = [] # Lista över besökare, max 3 privat
```

Metoder för att hantera djur

Nästa steg var att implementera metoder för att hantera djuren i parken:

- **add_animal(self, animal):**
Lägger till ett djur i listan self.animals. Jag använde append() för att enkelt lägga till ett djur.
- **list_animals(self, detailed=False):**
Returnerar en lista över djuren. Jag lade till ett argument detailed för att visa mer information om djuren när det behövs.

```
def add_animal(self, animal): 7 usages (3 dynamic) 🧑 Robin
    """Lägger till ett djur i djurparken."""
    self.__animals.append(animal)

def list_animals(self, detailed=False): 6 usages (6 dynamic) 🧑 Robin
    """Returnerar en lista med förenklad eller detaljerad information om djuren."""
    if detailed:
        return [str(animal) for animal in self.__animals] # Full info med __str__
    else:
        return self.__animals.copy() # Returnera djurobjekt
```

För att visa biljettpriser och tillval skapade jag:

- **show_ticket_price(self):** Returnerar biljettpriset i en formaterad sträng.
- **show_opening_hours(self):** Returnerar parkens öppettider.
- **show_addon_prices(self):** Returnerar tillvalen och deras priser som en formaterad lista.

```
def show_ticket_price(self) -> str: 1 usage (1 dynamic) 🧑 Robin
    """Returnerar biljettpris."""
    return f"Biljettpris: {self.__ticket_price} SEK."

def show_opening_hours(self) -> str: 1 usage (1 dynamic) 🧑 Robin
    """Returnerar öppettider för djurparken."""
    return f"Öppettider: {self.__opening_hours}."

def show_addon_prices(self) -> str: 1 usage (1 dynamic) 🧑 Robin
    """Visar tillgängliga tillval och deras priser."""
    return "\n".join(f"{addon}: {price} SEK" for addon, price in self._addons.items())
```

För att hantera besökare lade jag till:

- **add_visitor(self, visitor):** Lägger till en besökare om maxgränsen på 3 inte är nådd.
- **list_visitors(self):** Returnerar en lista över besökare som strängar.
- **remove_visitor(self, name):** Tar bort en specifik besökare från listan baserat på namn.

```
def add_visitor(self, visitor) -> str: 1 usage (1 dynamic) 🧑 Robin
    """Lägger till en besökare om maxgränsen inte är nådd."""
    if len(self._visitors) < 3:
        self._visitors.append(visitor)
        return f"Besökaren {visitor.name} har lagts till."
    return "Maxgränsen för 3 besökare är nådd."


def list_visitors(self) -> list: 1 usage (1 dynamic) 🧑 Robin
    """Returnerar en lista över besökare som strängar."""
    return [str(visitor) for visitor in self._visitors] if self._visitors else []

def remove_visitor(self, name: str) -> str: 1 usage (1 dynamic) 🧑 Robin
    """Tar bort en specifik besökare från listan."""
    for visitor in self._visitors:
        if visitor.name.lower() == name.lower():
            self._visitors.remove(visitor)
            return f"Besökaren {name} har tagits bort."
    return f"Ingen besökare med namnet {name} hittades."
```

Jag implementerade **interact_with_animal(self, name)** som låter besökaren interagera med ett specifikt djur genom att anropa djurets `interact()`-metod.

```
def interact_with_animal(self, name): 1 usage (1 dynamic) 🧑 Robin
    for animal in self.__animals:
        if animal.name == name:
            return animal.interact()
    return f"Inget djur med {name} hittat."
```


Slutligen lade jag till en `__str__`-metod för att enkelt kunna skriva ut information om djurparken.

```
def __str__(self):  Robin
    """Returnerar en strängrepresentation av djurparken."""
    return (
        f"Djurparken: {self._visitors}\n"
        f"Öppettider: {self.__opening_hours}\n"
        f"Biljettpris: {self.__ticket_price} SEK\n"
        f"Antal djur: {len(self.__animals)}\n"
        f"Antal besökare: {len(self._visitors)}"
    )
```

Utvecklingen av **Zoo-klassen** byggde på att skapa en central hantering för djur, besökare och parkens information. Genom att använda privata och skyddade attribut samt tydligt definierade metoder kunde jag säkerställa att klassen var flexibel och robust. Den fungerade som en **central hub** som kopplar samman hela programmet.

Animal-klassen

För att skapa en tydlig och återanvändbar struktur för alla djur i programmet utvecklade jag **Animal** som en abstrakt basklass. Genom att använda **abstrakta metoder** säkerställde jag att alla djur som ärvde från Animal-klassen implementerar specifika funktioner, samtidigt som gemensam funktionalitet hanterades i basklassen.

I konstruktorn definierade jag följande attribut som alla djur delar:

- `name`: Djurets namn.
- `age`: Djurets ålder.
- `favorite_food`: Djurets favoritmat.
- `hungry`: En bool som spårar om djuret är hungrigt eller mätt.
- `image_path`: Vägen till djurets bild.

```
class Animal(ABC):  # Robin +1
    def __init__(self, name: str, age: int, favorite_food: str, image_path: str):  # Robin +1
        """Initierar ett djur."""

        self.name = name
        self.age = age
        self.favorite_food = favorite_food
        self.hungry = True
        self.image_path = image_path
```

För att säkerställa att alla subklasser (som Lion, Elephant och Giraffe) implementerar specifik funktionalitet definierade jag två abstrakta metoder:

- **`interact()`**: En metod som definierar hur djuret interagerar med användaren.
- **`eat(food: str)`**: En metod som hanterar djurets matbeteende.

Dessa måste implementeras i varje subklass.

```
from abc import ABC, abstractmethod
```

```
@abstractmethod 6 usages (6 dynamic) 🧑 Robin +1
def interact(self) -> str:
    """Interagerar med djuret. Måste implementeras av subclasser."""
    pass

@abstractmethod 7 usages (7 dynamic) 🧑 rRoxn +1
def eat(self, food: str) -> str:
    """Funktion för att mata djuren. Måste implementeras av subclasser."""
    pass
```

Utöver de abstrakta metoderna lade jag till flera gemensamma metoder som hanterar djurens tillstånd och validering:

1. **toggle_hunger():**
En metod som växlar djurets hungerstatus mellan hungrig och mätt.
2. **is_hungry():**
Returnerar om djuret är hungrigt eller inte som en bool.
3. **validate_food(food: str):**
Kontrollerar om maten som ges är djurets favoritmat. Detta underlättar implementationen av eat() i subclasserna.

```
def toggle_hunger(self): 🧑 Robin
    """Byter hungerstatus på djuret."""
    self.hungry = not self.hungry

def is_hungry(self) -> bool: 🧑 Robin
    """Kontrollerar om djuret är hungrigt."""
    return self.hungry

def validate_food(self, food: str) -> bool: 🧑 Robin
    """Validerar om given mat är djurets favoritmat."""
    return food.lower() == self.favorite_food.lower()
```

Returnerar en strängrepresentation av djuret, med dess art, namn, ålder, favoritmat och hungerstatus.

(Notera att `get_species()` är en metod som varje subklass måste implementera för att returnera djurets art.)

```
def __str__(self): 🧑 Robin
    """Returnerar en strängrepresentation av djuret."""
    return (
        f"Art: {self.get_species()}, Namn: {self.name}, "
        f"Ålder: {self.age}, Favoritmat: {self.favorite_food}, "
        f"Hungrig: {'Ja' if self.hungry else 'Nej'}"
    )
```

Basklassen **Animal** skapades som en solid grund för att hantera djurens gemensamma attribut och funktioner. Abstrakta metoder tvingade varje subklass att implementera sina egna versioner av `eat()` och `interact()`, medan gemensamma metoder som `toggle_hunger()` och `validate_food()` förenklade logiken.

Nästa steg var att skapa specifika djurklasser, som **Lion**, där jag implementerade dessa metoder för att ge djuret sina unika beteenden.

Sub-klass Lion

Efter att ha byggt basklassen **Animal** utvecklade jag den första subklassen – **Lion**. Lejonet implementerade djurspecifika beteenden för att äta och interagera, vilket också fungerade som en testmodell för att säkerställa att Animal-basklassen fungerade korrekt.

Konstruktorn för **Lion** skickar vidare attributen till basklassen med hjälp av `super()`. Här definierade jag "Kött" som favoritmat för alla lejon.

```
class Lion(Animal): | 3 usages  👤Robin +1 *  
    def __init__(self, name: str, age: int, image_path: str):  👤Robin +1 *  
        """Initierar ett lejon."""  
        super().__init__(name, age, "Kött", image_path)
```

Jag implementerade `interact()`-metoden som specificerar vad som händer när användaren interagerar med lejonet. Lejonet blir hungrigt efter interaktionen, vilket gör det realistiskt och dynamiskt.

```
def interact(self) -> str: 6 usages (6 dynamic)  👤Robin +1  
    """Interaktion för lejon."""  
    self.hungry = True # Interaktion gör lejonet hungrigt  
    return f"{self.name} tittar stolt på dig."
```

Metoden `eat()` hanterar hur lejonet äter:

- Om lejonet inte är hungrigt returneras ett meddelande om detta.
- Om maten matchar favoritmaten används basklassens metod `toggle_hunger()` för att ändra hungerstatus.
- Om maten inte är favoritmat returneras ett meddelande att lejonet vägrar äta.

```
def eat(self, food: str) -> str: 7 usages (7 dynamic)  🧑 Robin +1
    """Lejonets specifika sätt att äta."""
    if not self.is_hungry(): # Använd basklassens metod för att kontrollera hunger
        return f"{self.name} är inte hungrig."
    if self.validate_food(food): # Kontrollera om det är favoritmat
        self.toggle_hunger() # Växla hungerstatus med basklassens metod
        return f"{self.name} sliter i sig {food} med en kraftig riv!"
    return f"{self.name} rynkar på nosen och vägrar äta {food}."
```

För att `Animal`-basklassens `__str__` ska fungera korrekt implementerade jag metoden `get_species()` som returnerar artens namn.

```
def get_species(self) -> str: 2 usages (2 dynamic)  🧑 Robin
    """Returnerar artens namn."""
    return "Lejon"
```

Första testet

För att verifiera att Animal- och Lion-klasserna fungerade korrekt:

1. Jag instansierade ett Lion-objekt i main.py och testade följande:
 - Att eat()-metoden fungerar som förväntat beroende på maten.
 - Att interact() ändrar hungerstatus och returnerar rätt meddelande.
 - Att `__str__` visar djurets information korrekt.

Jag skapade kod i min main.py:

Först skrev jag in imports för zoo, animal, och lion.

```
def main():  
    """Huvudmeny för att interagera med zoo."""  
    zoo = setup_zoo()  
    while True:  
        print("\nVälkommen till Amazing Zoo!")  
        print("1. Visa alla djur")  
        print("2. Mata ett djur")  
        print("3. Interagera med ett djur")  
        print("4. Visa biljettpris och tillval")  
        print("5. Avsluta")  
  
        choice = input("Välj ett alternativ: ")
```

Skrev en loop för olika val.

Sen skrev jag if-satser

```
if choice == "1":
    animals = zoo.list_animals(detailed=True)
    print("\nDjur i djurparken:")
    for animal in animals:
        print(animal)

elif choice == "2":
    name = input("Ange namnet på djuret du vill mata: ")
    food = input("Ange vilken mat du vill ge: ")
    print(zoo.feed_animal(name, food))

elif choice == "3":
    name = input("Ange namnet på djuret du vill interagera med: ")
    print(zoo.interact_with_animal(name))

elif choice == "4":
    print(zoo.show_ticket_price())
    print("Tillval:")
    print(zoo.show_addon_prices())

elif choice == "5":
    print("Tack för besöket!")
    break

else:
    print("Ogiltigt val, försök igen.")
```

Detta var då utdatan från min TUI meny.

Välkommen till Amazing Zoo!

1. Visa alla djur
2. Mata ett djur
3. Interagera med ett djur
4. Visa biljettpris och tillval
5. Avsluta

Välj ett alternativ: 1

Djur i djurparken:

Art: Lejon, Namn: Simba, Ålder: 5, Favoritmat: Kött, Hungrig: Ja

Välj ett alternativ: 2

Ange namnet på djuret du vill mata: Simba

Ange vilken mat du vill ge: Kött

Simba sliter i sig Kött med en kraftig riv!

Välj ett alternativ: 3

Ange namnet på djuret du vill interagera med: Simba

Simba tittar stolt på dig.

Genom att skapa **main.py** med en textbaserad meny (TUI) kunde jag snabbt testa alla viktiga funktioner i programmet manuellt. Detta gav mig en bra grund för att:

1. Säkerställa att Zoo och Lion fungerade som förväntat.
2. Identifiera eventuella buggar eller förbättringsmöjligheter.

När jag hade verifierat att allt fungerade i TUI-miljön gick jag vidare till att skriva **testfiler** med pytest för att automatisera testningen av alla klasser och metoder i projektet.

Utveckling av fler Subklasser: Giraffe, Elephant och LionCub

Efter att ha implementerat och testat **Lion**-klassen utökade jag projektet med fler subklasser för att representera andra djur i djurparken. Varje klass implementerade de abstrakta metoderna `eat()` och `interact()` från **Animal-basklassen**, men med unika beteenden.

Giraffe-klassen:

Denna klass är just precis som Lion-klassen, förutom att den ändrar beteende i `interact` och `eat` funktionerna.

```
from animals.animal import Animal

class Giraffe(Animal): 3 usages  🧑Robin +1 *
    def __init__(self, name: str, age: int, image_path: str):  🧑Robin +1 *
        """Initierar en giraff."""
        super().__init__(name, age, "Blad", image_path)

    def interact(self) -> str: 6 usages (6 dynamic)  🧑Robin
        """Interaktion för giraff."""
        self.hungry = True # Giraffen blir hungrig efter interaktion
        return f"{self.name} sträcker sig efter blad."

    def eat(self, food: str) -> str: 7 usages (7 dynamic)  🧑Robin +1
        """Giraffens specifika sätt att äta."""
        if not self.is_hungry(): # Använd basklassens metod
            return f"{self.name} är inte hungrig."
        if self.validate_food(food): # Kontrollera favoritmat
            self.toggle_hunger() # Byt hungerstatus
            return f"{self.name} sträcker sin långa hals för att nå {food}."
        return f"{self.name} ignorerar {food} och letar efter blad istället."

    def get_species(self) -> str: 2 usages (2 dynamic)  🧑Robin
        """Returnerar artens namn."""
        return "Giraff"
```

Elephant-klassen:

Denna klass är just precis som Lion-klassen, förutom att den ändrar beetende i interact och eat funktionerna.

```
from animals.animal import Animal

class Elephant(Animal): 3 usages  🧑Robin +1 *
    def __init__(self, name: str, age: int, image_path: str):  🧑Robin *
        """Initierar en elefant."""
        super().__init__(name, age, "jordnötter", image_path)

    def interact(self) -> str: 6 usages (6 dynamic)  🧑Robin +1
        """Interaktion för elefant."""
        self.hungry = True # Elefanten blir hungrig efter interaktion
        return f"{self.name} låter dig klappa dess snabel."

    def eat(self, food: str) -> str: 7 usages (7 dynamic)  🧑Robin +1
        """Elefantens specifika sätt att äta."""
        if not self.is_hungry(): # Använd basklassens metod
            return f"{self.name} är inte hungrig."
        if self.validate_food(food): # Kontrollera favoritmat
            self.toggle_hunger() # Byt hungerstatus
            return f"{self.name} plockar upp {food} med snabeln och äter det långsamt."
        return f"{self.name} trumpetar högt och vägrar äta {food}."

    def get_species(self) -> str: 2 usages (2 dynamic)  🧑Robin
        """Returnerar artens namn."""
        return "Elefant"
```

LionCub-klassen:

Efter att ha skapat **Lion**-klassen utökade jag funktionaliteten genom att skapa en ny subklass **LionCub**. Lejonungen (LionCub) ärver från **Lion** men introducerar ny logik som är anpassad för en yngre, lekfull version av lejonet.

I konstruktorn skickade jag vidare namn och bildväg till **Lion**-klassen med åldern satt till 0, eftersom **månader** används för att representera åldern istället.

Jag lade till ett nytt attribut, months, för att spara lejonungens ålder i månader.

```
from animals.lion import Lion

class LionCub(Lion): 4 usages  🧑Robin +1
    def __init__(self, name: str, months: int, image_path: str):  🧑Robin +1
        """Initierar en lejonunge."""
        super().__init__(name, age=0, image_path=image_path) # Ålder sätts till 0 eftersom månader används
        self.months = months
```

Lejonungens ätbeteende är unikt:

- Den äter försiktigt medan den leker med maten.
- Om det inte är favoritmat returneras ett meddelande att lejonungen hellre vill leka.
- Jag återanvände basklassens metoder `is_hungry()`, `validate_food()` och `toggle_hunger()`.

```
def eat(self, food: str) -> str: 7 usages (7 dynamic)  🧑Robin +1
    """Lejonungens specifika sätt att äta."""
    if not self.is_hungry(): # Använd basklassens metod
        return f"{self.name} är inte hungrig."
    if self.validate_food(food): # Kontrollera favoritmat
        self.toggle_hunger() # Byt hungerstatus
        return f"{self.name} tuggar försiktigt på {food} medan den leker."
    return f"{self.name} vägrar äta {food} och försöker leka istället."
```

Jag lade till en metod som returnerar lejonungens ålder i månader, vilket skiljer sig från andra djurklasser som använder år för ålder.

```
def get_age_in_month(self) -> int: 1 usage 🧑 Robin
    """Returnerar åldern i månader."""
    return self.months
```

Interaktionen med lejonungen är lekfull. Den "nafsar på ditt finger" och blir hungrig efter interaktionen.

```
def interact(self) -> str: 6 usages (6 dynamic) 🧑 Robin
    """Interaktion för lejonungen."""
    self.hungry = True
    return f"{self.name} nafsar på ditt finger."
```

Denna metod returnerar artens namn, vilket behövs för att basklassens `__str__` ska fungera korrekt.

```
def get_species(self) -> str: 3 usages (2 dynamic) 🧑 Robin
    """Returnerar artens namn."""
    return "Lejon"
```

LionCub är en utökad version av **Lion** som har extra logik för ålder i månader och ett mer lekfullt beteende:

Jag har då lagt till denna `get_species` funktionen för att komplimentera för `__str__` funktionen i `animal.py`. Så att detta ändra utskriften av arten till svenska.

Automatiserade tester med pytest

Efter att ha utvecklat klasserna för djuren och djurparken var nästa steg att säkerställa att alla funktioner fungerade korrekt. För att effektivt testa koden skrev jag en testfil kallad **test_zoo.py** och använde **pytest** som testverktyg. Automatiserade tester gjorde det möjligt att snabbt identifiera fel och bekräfta att alla delar av programmet fungerade som förväntat.

För att göra testerna organiserade och återanvändbara använde jag **pytest-fixturer**. Dessa fixturer skapade instanser av djur, zoo och besökare som användes i flera tester. Nedan så ser man då vad för fixturer jag har skapat. Detta är då fixturer som pytest skapar för att stimulera tester med data.

Fixturerna gjorde att jag inte behövde duplicera kod för att skapa objekt i varje test.

```
# Fixturer för djuren
@pytest.fixture 9 usages  🧑 rRoxn
def create_lion():
    return Lion("Simba", 5, "assets/images/lion.png")

@pytest.fixture 9 usages  🧑 rRoxn
def create_elephant():
    💡 return Elephant("Dumbo", 10, "assets/images/elephant.png")

@pytest.fixture 9 usages  🧑 rRoxn
def create_giraffe():
    return Giraffe("Melman", 7, "assets/images/giraffe.png")

@pytest.fixture 9 usages  🧑 Robin +1
def create_lioncub():
    return LionCub("Nala", 14, "assets/images/lioncub.jpeg")
```



```
# Fixturer för Zoo och Visitor
@pytest.fixture 6 usages 🧑 Robin
def create_zoo():
    zoo = Zoo("Djurparken", "09:00-18:00", 100, {"Mata djur": 50, "Detaljerad info": 30})
    zoo.add_animal(Lion("Simba", 5, "assets/images/lion.png"))
    zoo.add_animal(Elephant("Dumbo", 10, "assets/images/elephant.png"))
    zoo.add_animal(Giraffe("Melman", 7, "assets/images/giraffe.png"))
    return zoo

@pytest.fixture 9 usages 🧑 Robin
def create_visitor():
    return Visitor("Anna", 300)
```

Jag testade djurens eat()-metoder för att säkerställa att de:

- Åt sin favoritmat korrekt.
- Vägrade mat som inte var deras favorit.
- Uppdaterade hungerstatus som förväntat.

```
# Tester för eat
def test_lion_eat(create_lion): 🧑 Robin
    create_lion.hungry = True
    assert create_lion.eat("kött") == "Simba sliter i sig kött med en kraftig riv!"
    assert create_lion.hungry is False

def test_elephant_eat(create_elephant): 🧑 Robin
    create_elephant.hungry = True
    assert create_elephant.eat("jordnötter") == "Dumbo plockar upp jordnötter med snabeln och äter det långsamt."
    assert create_elephant.hungry is False

def test_giraffe_eat(create_giraffe): 🧑 Robin
    create_giraffe.hungry = True
    assert create_giraffe.eat("blad") == "Melman sträcker sin långa hals för att nå blad."
    assert create_giraffe.hungry is False

def test_lioncub_eat(create_lioncub): 🧑 Robin
    create_lioncub.hungry = True
    assert create_lioncub.eat("kött") == "Nala tuggar försiktigt på kött medan den leker."
    assert create_lioncub.hungry is False
```

Jag verifierade att djurens interact()-metoder returnerade rätt meddelanden och att hungerstatus ändrades till **True** efter interaktionen.

```
# Tester för interact
def test_lion_interact(create_lion):  🧑 Robin
    assert create_lion.interact() == "Simba tittar stolt på dig."
    assert create_lion.hungry is True

def test_elephant_interact(create_elephant):  🧑 Robin
    assert create_elephant.interact() == "Dumbo låter dig klappa dess snabel."
    assert create_elephant.hungry is True

def test_giraffe_interact(create_giraffe):  🧑 Robin
    assert create_giraffe.interact() == "Melman sträcker sig efter blad."
    assert create_giraffe.hungry is True

def test_lioncub_interact(create_lioncub):  🧑 Robin
    assert create_lioncub.interact() == "Nala nafsar på ditt finger."
    assert create_lioncub.hungry is True
```

För att säkerställa att bildvägarna var korrekt inställda för varje djur skrev jag specifika tester.

```
# Tester för bildväg
def test_lion_image_path(create_lion):  🧑 Robin
    assert create_lion.image_path == "assets/images/lion.png"

def test_elephant_image_path(create_elephant):  🧑 Robin
    assert create_elephant.image_path == "assets/images/elephant.png"

def test_giraffe_image_path(create_giraffe):  🧑 Robin
    assert create_giraffe.image_path == "assets/images/giraffe.png"

def test_lioncub_image_path(create_lioncub):  🧑 Robin
    assert create_lioncub.image_path == "assets/images/lioncub.jpeg"
```

Jag testade Zoo-klassens funktioner, som att lista djur, mata djur och interagera med djur. Detta säkerställde att Zoo-klassen fungerade som en central hub.

```
def test_zoo_feed_animal(create_zoo): 👤Robin
    response = create_zoo.feed_animal("Simba", "kött")
    assert response == "Simba sliter i sig kött med en kraftig riv!"

def test_zoo_interact_with_animal(create_zoo): 👤Robin
    response = create_zoo.interact_with_animal("Dumbo")
    assert response == "Dumbo låter dig klappa dess snabel."
```

Jag skrev tester för att verifiera att besökarens budget och kundvagn fungerade korrekt när de lade till och köpte tillval.

```
# Tester för Visitor

def test_visitor_add_to_cart(create_visitor): 👤Robin
    create_visitor.add_to_cart("Mata djur", 50)
    assert len(create_visitor.cart) == 1
    assert create_visitor.budget == 250

def test_visitor_purchase(create_visitor): 👤Robin
    create_visitor.add_to_cart("Mata djur", 50)
    create_visitor.purchase()
    assert create_visitor.budget == 250
    assert len(create_visitor.cart) == 0
```


Vad gör testen?

Setup med Fixtur:

- **create_lion** är en **pytest-fixtur** som skapar en instans av ett lejon (Lion-klassen) med namnet "Simba", ålder 5 och en bildväg.
- **create_lion.hungry = True** säkerställer att lejonet är hungrigt innan testet börjar.
- **Anrop av Metoden:**
- **create_lion.eat("kött")** anropar eat()-metoden från **Lion-klassen** med mat som argument ("kött").
- Lejonet ska nu:
 - Kontrollera att det är hungrigt.
 - Kontrollera om "kött" är dess favoritmat.
 - Uppdatera hungerstatus till **False** (mätt) om det äter sin favoritmat.
- **Användning av assert:**
- **assert** används för att kontrollera om värdet som returneras eller ändras är det förväntade. Om det inte är så, kastar testet ett **fel**.
- **assert create_lion.eat("kött") == "Simba sliter i sig kött med en kraftig riv!"**
 - Kontrollerar att metoden returnerar exakt rätt sträng.
- **assert create_lion.hungry is False**
 - Kontrollerar att hungerstatus har ändrats till **False** efter att lejonet ätit.

Detta är en kort beskrivning av ett test så ni förstår vad som händer.

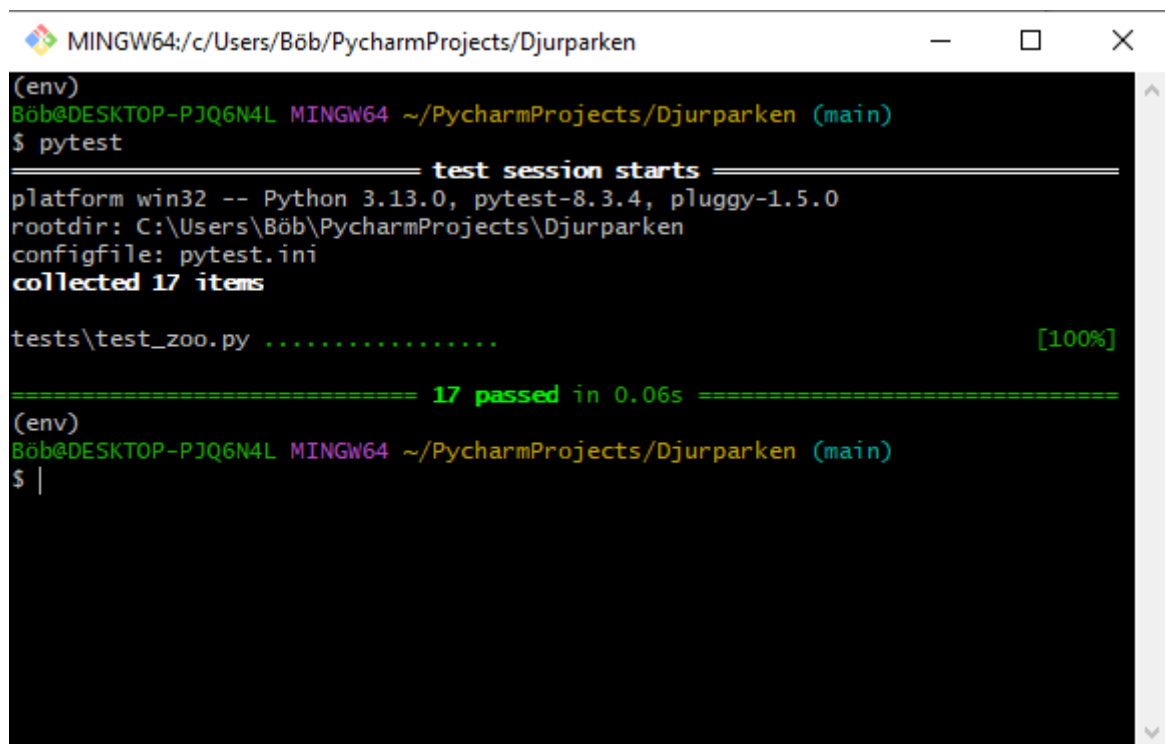
För att köra detta test efter vi skrivit vår testkod är genom att starta en terminal.

Börjar med att aktivera vår virtuella miljö.

Source env/Scripts/activate

Detta kommando kan skilja sig hur dina mappar ser ut.

Sedan skriver du bara pytest, detta är ju redan installerat sedan tidigare.

A screenshot of a Windows terminal window titled 'MINGW64:/c/Users/Böb/PycharmProjects/Djurparken'. The terminal shows a successful pytest execution. The prompt is '(env) Böb@DESKTOP-PJQ6N4L MINGW64 ~/PycharmProjects/Djurparken (main)'. The user enters '\$ pytest'. The output shows 'test session starts', platform and version information, root directory, config file, and 'collected 17 items'. A progress bar for 'tests\test_zoo.py' is shown with green dots and '[100%]' in green. The final result is '17 passed in 0.06s' in green, flanked by equals signs. The prompt returns to '\$ |'.

Här ser man då att 17 av testerna passerade. Det var ju alla så det blir grönt och 100%. Under utvecklingen fick jag ju massa error, och syntax fel. Detta är ju de som är snabbt och bra med pytest är att du slipper köra om koden flera gånger för att testa ny saker under utvecklingen.

Nu när alla våra funktioner fungerar så fortsätter vi och börjar att utveckla vårt GUI med tkinter.

