Eksempelløsning til en GUI med en tilsluttet SQL-database

# Introduktion

I dette dokument vil du opbygge en eksempel-løsning til et program med en GUI og en tilsluttet SQL-database. Dette dokument beskriver i detaljer de enkelte trin, som programmet er bygget op i. Den indeholder også den kode, der skal kopieres ind.

Formålet med dette dokument er at give dig mulighed for at anvende den viden, du har fået om GUI og SQL-databaser i de sidste øvelser, på et lidt større program og for at få mere øvelse. Desuden vil denne opgave give dig en idé om, hvad du kan forvente til din eksamen.

For database og ORM bruger du din viden fra opgaverne \05\_sql\_database\S2200\_db\_ osv...

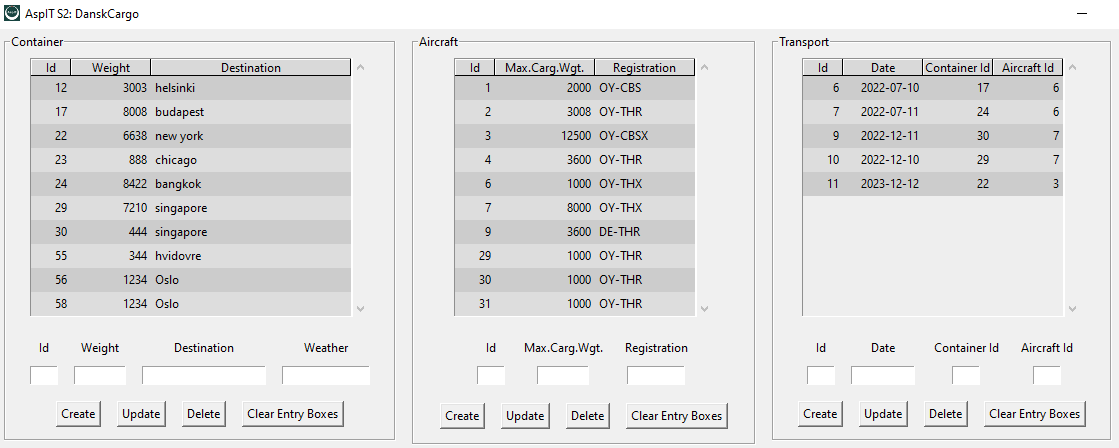
Du har allerede bygget en stor del af GUI'en i de tidligere GUI-opgaver. (Især i \04\_gui\S2040\_gui\_exercise4.py).

Åbn nu filen \06\_danskcargo\DanskCargo Eksamensopgave Python.docx i opgave-github-repositoriumet. Dette er en rigtig S2-eksamensopgave. I forhold til originalen er der kun foretaget mindre tilpasninger, fordi den oprindelige eksamen dengang skulle programmeres med C# i stedet for Python.

Læs først hele eksamensopgaven grundigt igennem. Så løser du denne opgave trin for trin i dette dokument.

Upload dine løsninger til GitHub mindst én gang om dagen. Gerne oftere, hvis du vil. På den måde kan din lærer følge dine fremskridt og støtte dig. Det giver dig også adgang til en sikkerhedskopi, hvis noget går galt.

Den færdige GUI vil se nogenlunde sådan her ud:



# Lagdeling

Dette program bliver større end dine tidligere programmer. Derfor fordeler du programkoden over flere filer. For at gøre dette bruger du princippet om den såkaldte lagdeling. Her fordeler du programkoden på flere filer eller moduler. Der bør være så lidt interaktion som muligt mellem modulerne. Det hjælper med at holde programmet overskueligt. Det gør det også lettere at ændre programmet senere, fordi ændringer i et modul ikke påvirker andre moduler, eller kun få steder.

Opret en ny mappe kaldet danskcargo i pycharm i dit løsningsprojekt.

Opret nu fire indledningsvis tomme filer i denne mappe:

* danskcargo\_data.py
* danskcargo\_func.py
* danskcargo\_sql.py
* danskcargo\_gui.py
* I danskcargo\_data.py definerer du databasens datastruktur.
* danskcargo\_func.py indeholder det såkaldte funktionelle lag. Det drejer sig for det meste om funktioner, der er baseret på indholdet af den konkrete projekt.
* I danskcargo\_sql.py skriver du alle funktioner, der interagerer med SQL-databasen.
* danskcargo\_gui.py skal indeholde alle GUI-funktioner.

# Milepæl 1: Læsning fra database, GUI til container

I dette kapitel opretter vi GUI'en til Container-klassen, opretter databasen med nogle testdata og læser disse data ind i GUI'en.

## danskcargo\_data.py

I denne fil definerer du databasens datastruktur. Først importerer du nogle objekter fra sqlalchemy-bibliotekerne.

Kopier de følgende kodelinjer i danskcargo\_data.py :

from sqlalchemy.orm import declarative\_base

from sqlalchemy import Column

from sqlalchemy import String, Integer

Hvis du endnu ikke har installeret sqlalchemy, skal du gøre det nu. For at gøre dette skal du åbne et terminalvindue i Pycharm og udføre denne kommando: "pip install SQLAlchemy".

Nu definerer du en klasse Base, som skal danne grundlag for dine egne klasser, hvori du definerer datastrukturer for databasen og dit Python-program på samme tid. Dette er et godt eksempel på, hvor nyttig nedarvning kan være i objektorienteret programmering. Med en enkelt linje kode får du en klasse med omfattende funktionalitet, som du kan overføre til dine egne klasser.

Base = declarative\_base() # creating the registry and declarative base classes - combined into one step. Base will serve as the base class for the ORM mapped classes we declare.

På basis af Base kan du nu definere din første egen klasse:

\_\_tablename\_\_ angiver navnet på tabellen i SQL-databasen.

Så definerer du nogle attributter i klassen, som også bliver kolonner i database-tabellen.

Integer og String er ikke Python-datatyper, men foruddefinerede klasser i sqlalchemy, som indeholder databasefunktionalitet. Du angiver desuden, at attributten id er primærnøglen for denne databasetabel.

class **Container**(Base):

\_\_tablename\_\_ = *"container"*

id = Column(Integer, primary\_key=True)

weight = Column(Integer)

destination = Column(String)

Definer en funktion \_\_repr\_\_(), så objekter af klassen Container kan udskrives i en læsbar form. Du bruger ikke denne funktion i det færdige program, men kun til test og fejlfinding.

def \_\_repr\_\_(self): # Optional. Only for test purposes.

return *f"Container(*{self.id=:*4*} {self.weight=:*5*} {self.destination=}*)"*

Snart vil du kunne se indholdet af databasen i vores GUI. I GUI-biblioteket tkinter bruger du widget Treeview til dette formål. Denne widget forventer dataposterne som tupler. Derfor er det praktisk at have en metode i klassen, som konverterer klassens objekter til en tupel.

def **convert\_to\_tuple**(self): # Convert Container to tuple

return self.id, self.weight, self.destination

Definer nu en funktion, der finder ud af, om et objekt i klassen indeholder gyldige data. Ugyldige data kan skyldes forkerte indtastninger, men også at et objekt er markeret som slettet ("soft deleted"). Vi vil diskutere sidstnævnte mere detaljeret senere.

Funktionen kontrollerer, om attributten kan omdannes til en heltalsværdi. For at forhindre, at programmet afbrydes med en fejlmeddelelse, hvis konverteringen mislykkes, placeres konverteringen i en try-blok. Hvis der opstår en fejl i en try-blok, afsluttes programmet ikke, men koden i except-blokken udføres. Mere om dette her: <https://pythonbasics.org/try-except/>.

def **valid**(self):

try:

value = int(self.weight)

except ValueError:

return False

return value >= 0

Som modstykke til den funktion, der konverterer klasseobjekter til en tupel, har du også brug for en funktion, der konverterer tupler til klasseobjekter. Denne funktion er en såkaldt statisk metode. Det betyder, at denne metode ikke anvendes på objekter i klassen (bemærk, at metoden ikke har nogen parameter "self"), men at den stadig tilhører klassen. Statiske metoder giver mening, når funktionen ikke anvendes på en instans af klassen, men stadig hører til klassen med hensyn til indhold. I dette tilfælde er det tydeligere, hvis funktionen er defineret i klassen og ikke et andet sted. Hvis du har lyst, kan du lære mere om statiske metoder her: <https://youtu.be/SXApHXsDe8I?t=38>.

Den første linje med @-tegnet kaldes en dekorator. Vi behandler ikke dette i dette modul. Men hvis du er nysgerrig, kan du læse mere om det her: <https://www.datacamp.com/tutorial/decorators-python>.

@staticmethod

def **convert\_from\_tuple**(tuple\_): # Convert tuple to Container

container = Container(id=tuple\_[0], weight=tuple\_[1], destination=tuple\_[2])

return container

## danskcargo\_func.py

Dette Python-script forbliver tomt i starten. Du har først brug for den, når du har programmeret klasser for fly og transporter ud over containerklassen og ønsker at kontrollere, om oplysningerne i de forskellige klasser passer sammen.

## danskcargo\_sql.py

Først importerer du objekter fra SQL-biblioteket sqlalchemy og din egen fil danskcargo\_data.py.

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, select

from danskcargo\_data import Container, Base

Sqlalchemy kan arbejde med mange forskellige databaser såsom Postgresql, MySQL, Oracle, MS-SQL, Firebird, Sybase. Vi bruger SQLite her, fordi denne database kræver ingen ekstra installation eller separate programmer. SQLite har dog en særlig egenskab: Foreign key constraints er som standard deaktiveret. Derfor har vi brug for følgende kode for at forhindre sletning af en post i SQLite, hvis denne post refereres i en anden tabel (via en fremmed nøgle):

# add the following 7 lines to make foreign key constraints work https://docs.sqlalchemy.org/en/14/dialects/sqlite.html#sqlite-foreign-keys

from sqlalchemy.engine import Engine

from sqlalchemy import event

@event.listens\_for(Engine, *"connect"*)

def **set\_sqlite\_pragma**(dbapi\_connection, connection\_record):

cursor = dbapi\_connection.cursor()

cursor.execute(*"PRAGMA foreign\_keys=ON"*)

cursor.close()

Ved hjælp af den globale variabel Database fortæller du sqlalchemy, hvilken typ SQL-database du bruger (her sqlite), hvor databasen er placeret, og hvad den hedder.

Database = *'sqlite:///danskcargo.db'* # first part: database type, second part: file path

For at teste databasen, før du er færdig med at skrive GUI'en, skal du oprette nogle testdata.

def **create\_test\_data**(): # Optional. Used to test data base functions before gui is ready.

with Session(engine) as session:

new\_items = []

new\_items.append(Container(weight=1200, destination=*"Oslo"*))

new\_items.append(Container(weight=700, destination=*"Helsinki"*))

new\_items.append(Container(weight=1800, destination=*"Helsinki"*))

new\_items.append(Container(weight=1000, destination=*"Helsinki"*))

session.add\_all(new\_items)

session.commit()

Denne funktion returnerer en liste med alle poster i en tabel i vores SQL-database. Her er classparam navnet på en af vores dataklasser (Container, Aircraft, Transport). Via attributten \_\_tablename\_\_ i dataklassen kender funktionen navnet på den pågældende tabel i SQL-databasen.

def **select\_all**(classparam): # https://docs.sqlalchemy.org/en/14/tutorial/data\_select.html

# return a list of all records in classparams table

with Session(engine) as session:

records = session.scalars(select(classparam)) # very useful for converting into our data class

result = []

for record in records:

# print(record)

result.append(record)

return result

Den følgende funktion fungerer på samme måde som den foregående funktion select\_all, men SQL-forespørgslen er desuden begrænset til et bestemt ID (her primærnøglen). Derfor returnerer funktionen kun én post.

def **get\_record**(classparam, record\_id): # https://docs.sqlalchemy.org/en/14/tutorial/data\_select.html

# return the record in classparams table with a certain id

with Session(engine) as session:

record = session.scalars(select(classparam).where(classparam.id == record\_id)).first() # very useful for converting into our data class

return record

Nu mangler kun hovedprogrammet i dette Python-script. If-betingelsen kontrollerer, om selve scriptet udføres eller kun importeres. Førstnævnte gælder kun, hvis vi ønsker at teste vores databasefunktioner. Ellers kører vi altid danskcargo\_gui.py, og vores andre scripts importeres kun. I så fald udføres Else-delen af det følgende hovedprogram.

if \_\_name\_\_ == *"\_\_main\_\_"*: # Executed when invoked directly

engine = create\_engine(Database, echo=False, future=True) # https://docs.sqlalchemy.org/en/14/tutorial/engine.html The start of any SQLAlchemy application is an object called the Engine. This object acts as a central source of connections to a particular database, providing both a factory as well as a holding space called a connection pool for these database connections. The engine is typically a global object created just once for a particular database server, and is configured using a URL string which will describe how it should connect to the database host or backend.

Base.metadata.create\_all(engine)

# create\_test\_data()

print(select\_all(Container))

print(get\_record(Container, 2))

else: # Executed when imported

engine = create\_engine(Database, echo=False, future=True) # https://docs.sqlalchemy.org/en/14/tutorial/engine.html The start of any SQLAlchemy application is an object called the Engine. This object acts as a central source of connections to a particular database, providing both a factory as well as a holding space called a connection pool for these database connections. The engine is typically a global object created just once for a particular database server, and is configured using a URL string which will describe how it should connect to the database host or backend.

Base.metadata.create\_all(engine)

Udfør danskcargo\_sql.py én gang, mens linjen create\_test\_data() ikke er kommenteret ud. Dette opretter databasen og fylder den med nogle testdata. Udkommenter denne linje bagefter.

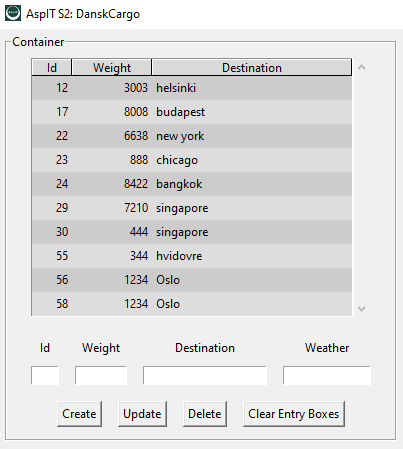
Kør nu scriptet et par gange mere, mens du leger med select\_all og get\_record-funktionerne i hovedprogrammet.

## danskcargo\_gui.py

Du har allerede bygget en stor del af GUI'en til denne opgave i din løsning til øvelse S2040\_gui\_exercise4.py. Du kan nu skrive GUI'en i danskcargo\_gui.py ved hjælp af en næsten identisk løsning.

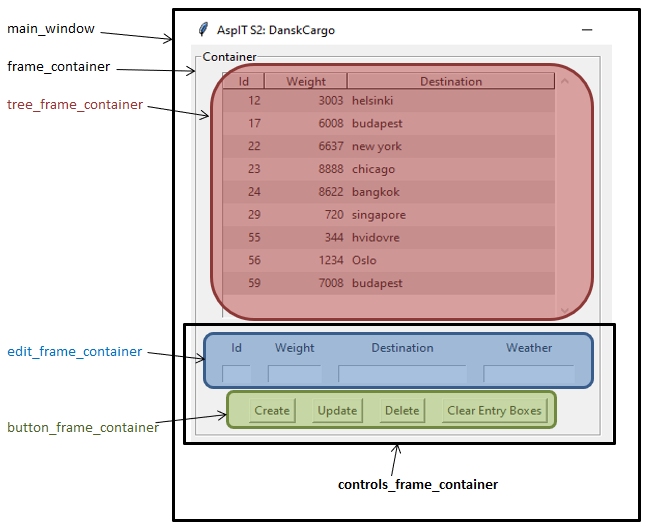
Før du kan programmere en GUI, skal du først vide, hvordan GUI'en skal se ud. Du har brug for en plan. Ofte laver man et første udkast med blyant og papir. Eller du er heldig og har en skabelon. Brug grafikken fra kapitlet "Introduktion" (øverst i dette dokument) som en skabelon.

For det første beskæftiger vi os kun med den venstre tredjedel af GUI'en, dvs. klassen Container. I dette afsnit er navnene på de tkinter-widgets, der anvendes, skrevet i parentes. Du kan se et programvindue (Tk) i den følgende grafik. Næsten hele programvinduet fyldes af en ramme med etiketten "Container" (LabelFrame). Indeni er der en liste med data (Treeview) med en rullebjælke. Længere nede er der nogle tekster (labels), hver med et indtastningsfelt (entry) under dem. Nederst er der nogle knapper (Button). På billedet ses ikke yderligere usynlige rammer, som vi kun bruger til at arrangere widgets mere pænt.

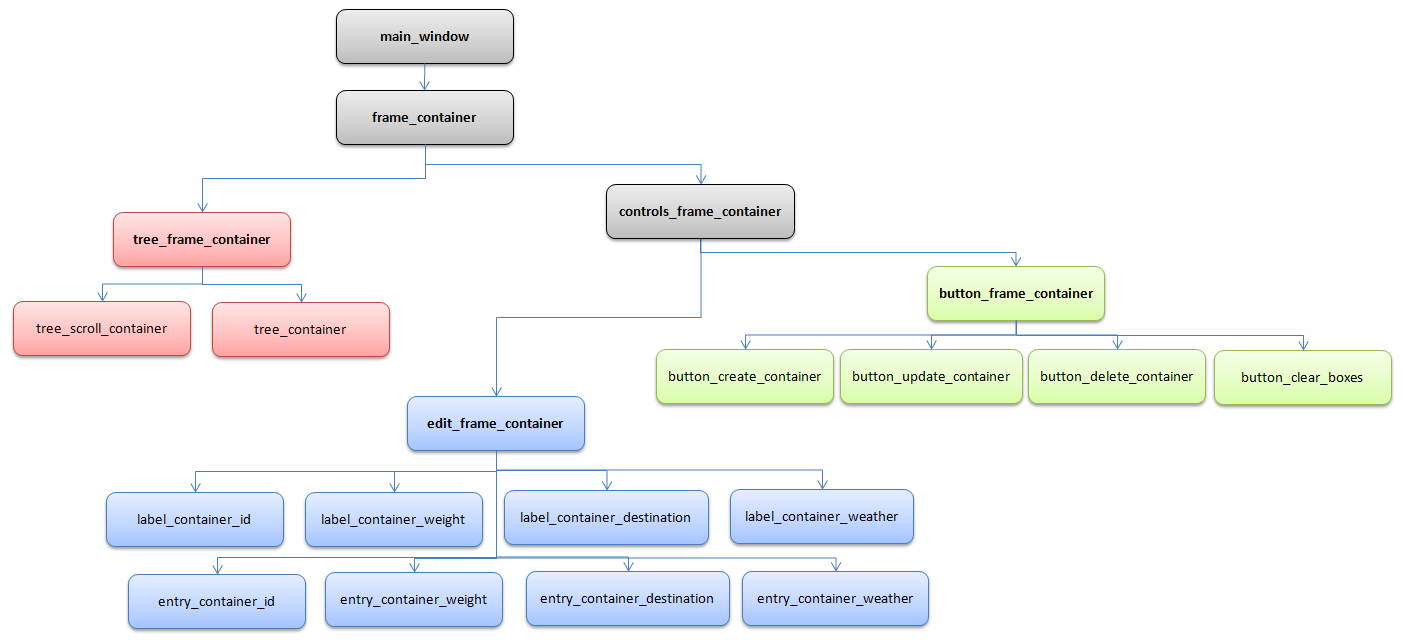


I S2000\_gui\_links.py finder du links til den detaljerede dokumentation for widgets. Du kan især læse i dokumentationen, hvilke konfigurationsmuligheder (ved hjælp af valgfrie parametre) de enkelte widgets har.

I den følgende grafik kan du se variabelnavne for de synlige og usynlige rammer, som du skal bruge i programkoden:



Her er en oversigt over navnene på alle de widgets, som vi bruger til containere. Tkinter er hierarkisk. Når du definerer en ny widget, skal du angive, hvilken overordnet widget den nye widget hører til. Du kan se dette hierarki og tilknytningerne i følgende grafiske fremstilling. Farverne i hierarkiet svarer til farverne i den foregående grafik.



Kopier de nedenstående stykker programmeringskode en efter en ind i danskcargo\_gui.py.

### Import

Først skal du importere GUI-bibliotekerne:

import tkinter as tk

from tkinter import ttk

Derudover importer dine egne filer:

import danskcargo\_data as dcd

import danskcargo\_sql as dcsql

Nu kan du bruge klasserne, funktionerne og variablerne fra de to andre filer i danskcargo\_gui.py ved at sætte de tilsvarende filnavne foran disse objekter. Nøgleordet "as" giver dig mulighed for at bruge forkortelserne "dcd” eller "dcsql" i stedet for at bruge de lange filnavne.

### Globale Variabler

Det er en god idé ikke at bruge såkaldte hårde værdier i programmeringskoden. Det er tal eller strings, som du måske ønsker at ændre senere.

I stedet skal du definere disse værdier som globale variabler et sted i vores kode, hvor det er let at finde dem. På denne måde forbliver din kode overskuelig, og det er meget nemt at ændre en af variablerne. Selv om den samme værdi bruges mange steder i programmet, behøver du kun at justere koden ét sted.

# region global constants

padx = 8 # Horizontal distance to neighboring objects

pady = 4 # Vertical distance to neighboring objects

rowheight = 24 # rowheight in treeview

treeview\_background = *"#eeeeee"* # color of background in treeview

treeview\_foreground = *"black"* # color of foreground in treeview

treeview\_selected = *"#206030"* # color of selected row in treeview

oddrow = *"#dddddd"* # color of odd row in treeview

evenrow = *"#cccccc"* # color of even row in treeview

# endregion global constants

Bemærk den første og sidste linje i kodeafsnittet (henholdsvis region og endregion). Disse linjer har ingen indflydelse på programmet, men de hjælper med at bevare overblikket. Ved at klikke på minustegnet til venstre for programkoden kan du lukke hele regionen sammen.

Kopier nu følgende linjer ind i danskcargo\_gui.py for at definere de resterende regioner.

# region container functions

# endregion container functions

# region common functions

# endregion common functions

# region common widgets

# endregion common widgets

# region container widgets

# endregion container widgets

# region main program

# endregion main program

Hvis du finder kodelinjer senere i dette kapitel (milepæl 1), der starter eller slutter en region, må du ikke kopiere disse linjer ind i dit Python-script. Disse linjer er der kun, så du ved, hvilken region du skal kopiere koden til.

### Fælles widgets

Hver tkinter GUI starter med et hovedvindue (Tk). Desuden kan du angive dens titel og størrelse.

# region common widgets

main\_window = tk.Tk() # Define the main window

main\_window.title(*'AspIT S2: DanskCargo'*) # Text shown in the top window bar

main\_window.geometry(*"500x500"*) # window size

Du kan vælge mellem foruddefinerede farveskemaer via Style-objektet.

style = ttk.Style() # Add style

style.theme\_use(*'default'*) # Pick theme

Definer farver og formatering af datalisterne (Treeview).

# Configure treeview colors and formatting. A treeview is an object that can contain a data table.

style.configure(*"Treeview"*, background=treeview\_background, foreground=treeview\_foreground, rowheight=rowheight, fieldbackground=treeview\_background)

style.map(*'Treeview'*, background=[(*'selected'*, treeview\_selected)]) # Define color of selected row in treeview

# endregion common widgets

### Hovedprogram, del 1

Nu har vi så meget kode i vores GUI, at vi kan afprøve den.

Det meste af tiden er hovedprogrammet i en GUI meget enkelt. Det er også tilfældet her: Den starter bare en uendelig løkke, der venter på brugerhandlinger (f.eks. museklik eller tekstinput).

Tilføj disse linjer nederst i danskcargo\_gui.py.

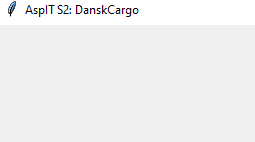
# region main program

if \_\_name\_\_ == *"\_\_main\_\_"*: # Executed when invoked directly. We use this so main\_window.mainloop() does not keep our unit tests from running.

main\_window.mainloop() # Wait for button clicks and act upon them

# endregion main program

Udfør nu danskcargo\_gui.py. Et tomt vindue skal åbnes. Resultatet bør se nogenlunde sådan ud:



### Widgets til containere

En LabelFrame er en ramme, der afbrydes på et sted af en tekst. Her defineres en LabelFrame, der skal indeholde alle vores objekter for dataklassen Container.

Som med enhver anden widget er det første, du gør, at angive, hvilket objekt den nye widget skal tilknyttes. (Her: main\_window)

Oftest indeholder widgetdefinitionen yderligere parametre, der beskriver den nye widget mere detaljeret. (Her: text="Container")

I den anden linje definerer du den nye widgets position i den overordnede widgets gitter (grid).

# region container widgets

# Define Labelframe which contains all container related GUI objects (data table, labels, buttons, ...)

frame\_container = tk.LabelFrame(main\_window, text=*"Container"*) # https://www.tutorialspoint.com/python/tk\_labelframe.htm

frame\_container.grid(row=0, column=0, padx=padx, pady=pady, sticky=tk.N) # https://www.tutorialspoint.com/python/tk\_grid.htm

Definer treeviewen og dens rullebjælke. Hvis du vil placere disse to objekter korrekt i forhold til hinanden, skal du sætte dem i en ramme og placere begge widgets ved hjælp af gitteret.

# Define data table (Treeview) and its scrollbar. Put them in a Frame.

tree\_frame\_container = tk.Frame(frame\_container) # https://www.tutorialspoint.com/python/tk\_frame.htm

tree\_frame\_container.grid(row=0, column=0, padx=padx, pady=pady)

tree\_scroll\_container = tk.Scrollbar(tree\_frame\_container)

tree\_scroll\_container.grid(row=0, column=1, padx=0, pady=pady, sticky=*'ns'*)

tree\_container = ttk.Treeview(tree\_frame\_container, yscrollcommand=tree\_scroll\_container.set, selectmode=*"browse"*) # https://docs.python.org/3/library/tkinter.ttk.html#treeview

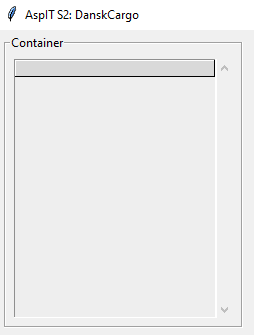
tree\_container.grid(row=0, column=0, padx=0, pady=pady)

tree\_scroll\_container.config(command=tree\_container.yview)

# endregion container widgets

Udfør nu danskcargo\_gui.py. Du bør se et tomt treeview med rullebjælker.

Resultatet bør se nogenlunde sådan ud:



Tilføj også de andre afsnit med programkode i dette afsnit "Widgets til containere" til regionen “container widgets”.

Definer nu kolonnerne og kolonneoverskrifterne i treeviewen. Desuden skal du definere to tags til baggrundsfarverne i treeviewen.

# Define the data table's formatting and content

tree\_container[*'columns'*] = (*"id"*, *"weight"*, *"destination"*) # Define columns

tree\_container.column(*"#0"*, width=0, stretch=tk.NO) # Format columns. Suppress the irritating first empty column.

tree\_container.column(*"id"*, anchor=tk.E, width=40) # "E" stands for East, meaning Right. Possible anchors are N, NE, E, SE, S, SW, W, NW and CENTER

tree\_container.column(*"weight"*, anchor=tk.E, width=80)

tree\_container.column(*"destination"*, anchor=tk.W, width=200)

tree\_container.heading(*"#0"*, text=*""*, anchor=tk.W) # Create column headings

tree\_container.heading(*"id"*, text=*"Id"*, anchor=tk.CENTER)

tree\_container.heading(*"weight"*, text=*"Weight"*, anchor=tk.CENTER)

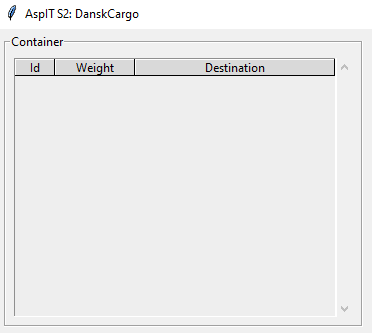
tree\_container.heading(*"destination"*, text=*"Destination"*, anchor=tk.CENTER)

tree\_container.tag\_configure(*'oddrow'*, background=oddrow) # Create tags for rows in 2 different colors

tree\_container.tag\_configure(*'evenrow'*, background=evenrow)

Udfør nu danskcargo\_gui.py. Nu har treeviewen markerede kolonner.

Resultatet bør se nogenlunde sådan ud:



Definer en usynlig ramme til at arrangere alle knapper, labels og entries. I grafikken ovenfor er det det blå og grønne område.

# Define Frame which contains labels, entries and buttons

controls\_frame\_container = tk.Frame(frame\_container)

controls\_frame\_container.grid(row=3, column=0, padx=padx, pady=pady)

For at kunne arrangere widgets mere pænt kombinerer vi alle labels og entries i en yderligere ramme. I grafikken ovenfor er det det blå område.

Indtastningswidgets i vores GUI hedder entry\_container\_id, entry\_container\_weight, entry\_container\_destination og entry\_container\_weather.

Det er en god idé at navngive alle widgets, så det er tydeligt, hvad de er, og hvordan de bruges. Det er bedst at bruge ensartede navngivningsregler for widgets i hele programmet.

I vores program er den første del af navnet widget-typen. Den anden del af navnet er dataklassen/SQL-tabellen. Den tredje del af navnet er klasseattributten/SQL-tabellens kolonne/felt.

Denne ensartede procedure vil gøre det meget lettere for os senere at kopiere programkoden for klassen Container og tilpasse den til de andre klasser Aircraft og Transport.

# Define Frame which contains labels (text fields) and entries (input fields)

edit\_frame\_container = tk.Frame(controls\_frame\_container) # Add tuple entry boxes

edit\_frame\_container.grid(row=0, column=0, padx=padx, pady=pady)

# label and entry for container id

label\_container\_id = tk.Label(edit\_frame\_container, text=*"Id"*) # https://www.tutorialspoint.com/python/tk\_label.htm

label\_container\_id.grid(row=0, column=0, padx=padx, pady=pady)

entry\_container\_id = tk.Entry(edit\_frame\_container, width=4, justify=*"right"*) # https://www.tutorialspoint.com/python/tk\_entry.htm

entry\_container\_id.grid(row=1, column=0, padx=padx, pady=pady)

# label and entry for container weight

label\_container\_weight = tk.Label(edit\_frame\_container, text=*"Weight"*)

label\_container\_weight.grid(row=0, column=1, padx=padx, pady=pady)

entry\_container\_weight = tk.Entry(edit\_frame\_container, width=8, justify=*"right"*)

entry\_container\_weight.grid(row=1, column=1, padx=padx, pady=pady)

# label and entry for container destination

label\_container\_destination = tk.Label(edit\_frame\_container, text=*"Destination"*)

label\_container\_destination.grid(row=0, column=2, padx=padx, pady=pady)

entry\_container\_destination = tk.Entry(edit\_frame\_container, width=20)

entry\_container\_destination.grid(row=1, column=2, padx=padx, pady=pady)

# label and entry for container destination

label\_container\_weather = tk.Label(edit\_frame\_container, text=*"Weather"*)

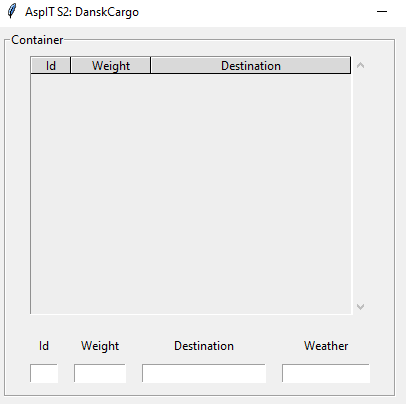
label\_container\_weather.grid(row=0, column=3, padx=padx, pady=pady)

entry\_container\_weather = tk.Entry(edit\_frame\_container, width=14)

entry\_container\_weather.grid(row=1, column=3, padx=padx, pady=pady)

Udfør nu danskcargo\_gui.py. Nedenfor kan du se labels og entries.

Resultatet skal se således ud:



Knapperne får også deres egen ramme, som er det grønne område i grafikken ovenfor.

# Define Frame which contains buttons

button\_frame\_container = tk.Frame(controls\_frame\_container)

button\_frame\_container.grid(row=1, column=0, padx=padx, pady=pady)

# Define buttons

button\_create\_container = tk.Button(button\_frame\_container, text=*"Create"*)

button\_create\_container.grid(row=0, column=1, padx=padx, pady=pady)

button\_update\_container = tk.Button(button\_frame\_container, text=*"Update"*)

button\_update\_container.grid(row=0, column=2, padx=padx, pady=pady)

button\_delete\_container = tk.Button(button\_frame\_container, text=*"Delete"*)

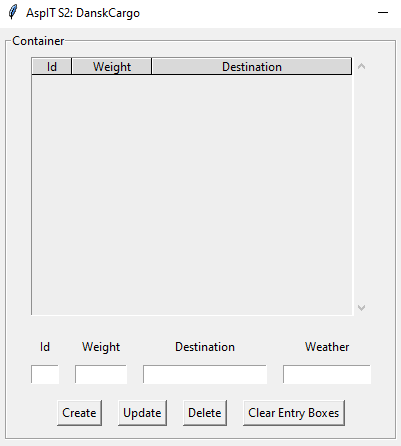
button\_delete\_container.grid(row=0, column=3, padx=padx, pady=pady)

button\_clear\_boxes = tk.Button(button\_frame\_container, text=*"Clear Entry Boxes"*)

button\_clear\_boxes.grid(row=0, column=4, padx=padx, pady=pady)

Udfør nu danskcargo\_gui.py. Du vil nu se knapperne nedenfor.

Resultatet skal se således ud:



Den synlige del af GUI'en for klassen Container er nu klar. Derefter skal du definere nogle funktioner, der skal udføres, når du interagerer med GUI'en.

### Funktioner til Entry Widgets

Først defineres de funktioner, der læser, tømmer og fylder dine indtastningswidgets med tekst.

# region container functions

def **read\_container\_entries**(): # Read content of entry boxes

return entry\_container\_id.get(), entry\_container\_weight.get(), entry\_container\_destination.get(),

def **clear\_container\_entries**(): # Clear entry boxes

entry\_container\_id.delete(0, tk.END) # Delete text in entry box, beginning with the first character (0) and ending with the last character (tk.END)

entry\_container\_weight.delete(0, tk.END)

entry\_container\_destination.delete(0, tk.END)

entry\_container\_weather.delete(0, tk.END)

def **write\_container\_entries**(values): # Fill entry boxes

entry\_container\_id.insert(0, values[0])

entry\_container\_weight.insert(0, values[1])

entry\_container\_destination.insert(0, values[2])

def **edit\_container**(event, tree): # Copy selected tuple into entry boxes. Parameter event is mandatory but we don't use it.

index\_selected = tree.focus() # Index of selected tuple

values = tree.item(index\_selected, *'values'*) # Values of selected tuple

clear\_container\_entries() # Clear entry boxes

write\_container\_entries(values) # Fill entry boxes

# endregion container functions

Den sidste funktion skriver indholdet af en udvalgt treeviewlinje ind i entry widgets.

Regionen indeholder navnet Container, fordi du senere vil skrive lignende funktioner for klasserne Aircraft og Transport.

### Link widgets med funktioner

Tilføj parameteren command til definitionen af knappen button\_clear\_boxes og tildel den funktionsnavnet clear\_container\_entries. På denne måde har du bestemt, at funktionen clear\_container\_entries skal udføres efter hvert klik med musen på denne knap.

button\_clear\_boxes = tk.Button(button\_frame\_container, text=*"Clear Entry Boxes"*, command=clear\_container\_entries)

Med metoden bind kan du for treeviews (og nogle andre widgets) definere, hvilken funktion der skal udføres ved en bestemt hændelse, f.eks. et museklik. Her ønsker vi at udføre funktionen edit\_container(), når der klikkes med musen på treeviewen. (Strengt taget, når museknappen slippes.) På denne måde kopieres dataene for denne linje til entry widgetterne efter et museklik på en linje i treeviewen.

Faktisk kan du kun give det rene funktionsnavn som parameter til metoden bind, præcis som vi lige har gjort med optionen “command” for knappen. For også at kunne give funktionsparametre (her især tree\_container) bruger vi et trick, nemlig lambda-funktionen. Vi behandler ikke lambda-funktioner nærmere i dette modul, men hvis du er nysgerrig, kan du læse mere om det her: <https://www.w3schools.com/python/python_lambda.asp>.

Tilføj følgende linje direkte under de andre linjer, hvor widget tree\_container er defineret:

tree\_container.bind(*"<ButtonRelease-1>"*, lambda event: edit\_container(event, tree\_container)) # Define function to be called, when an item is selected.

### Fælles funktioner

Tilføj følgende funktioner i regionen “common functions”.

Den første funktion læser alle containere fra databasen og skriver de gyldige (ikke slettede) containere ind i treeviewen. Ulige og lige rækker får forskellige baggrundsfarver. Bemærk, at funktionen ikke udtrykkeligt navngiver klassen Container. I stedet bruger vi en parameter class\_. På denne måde kan vi senere bruge den samme funktion også til klasserne Aircraft og Transport.

def **read\_table**(tree, class\_): # fill tree from database

count = 0 # Used to keep track of odd and even rows, because these will be colored differently.

result = dcsql.select\_all(class\_) # Read all containers from database

for record in result:

if record.valid(): # this condition excludes soft deleted records from being shown in the data table

if count % 2 == 0: # even

tree.insert(parent=*''*, index=*'end'*, iid=str(count), text=*''*, values=record.convert\_to\_tuple(), tags=(*'evenrow'*,)) # Insert one row into the data table

else: # odd

tree.insert(parent=*''*, index=*'end'*, iid=str(count), text=*''*, values=record.convert\_to\_tuple(), tags=(*'oddrow'*,)) # Insert one row into the data table

count += 1

Den anden funktion tømmer en dataliste (treeview).

# region common functions

def **empty\_treeview**(tree): # Clear treeview table

tree.delete(\*tree.get\_children())

Den tredje funktion bruger de to første funktioner til at opdatere en dataliste. Først tømmes datalisten og derefter læses den ind igen fra databasen. Ved også at gøre dataklassen til en funktionsparameter her kan vi bruge den samme funktion til forskellige klasser.

def **refresh\_treeview**(tree, class\_): # Refresh treeview table

empty\_treeview(tree) # Clear treeview table

read\_table(tree, class\_) # Fill treeview from database

# endregion common functions

### Hovedprogram, del 2

Tilføj en linje til hovedprogrammet, der initialiserer GUI'en ved at fylde treeviewen fra databasen.

Hovedprogrammet ser så således ud:

if \_\_name\_\_ == *"\_\_main\_\_"*: # Executed when invoked directly. We use this so main\_window.mainloop() does not keep our unit tests from running.

refresh\_treeview(tree\_container, dcd.Container) # Load data from database

main\_window.mainloop() # Wait for button clicks and act upon them

Kør programmet. Resultatet bør se nogenlunde sådan ud:



Klik på nogle linjer i treeviewen og prøv knappen "Clear Entry Boxes".

GUI'en er nu klar, du kan læse data fra databasen til dit treeview, du kan kopiere data fra treeviewen til entry widgets med et enkelt klik, og du kan slette indholdet af entry widgets med et enkelt klik.

Nu har du nået den første milepæl. Du kan sammenligne din løsning med filerne i mappen \milestones\danskcargo\10\_S2040\_gui\_exercise4.

# Milepæl 2: Create, update, delete i datenbasen

I dette kapitel skriver vi funktioner til oprettelse, opdatering og sletning af poster i databasen.

Alle ændringer i dette kapitel skal indarbejdes i filen danskcargo\_sql.py.

Parameteren record kan have enhver datatype, der er defineret i danskcargo\_data.py ved at arve fra basisklassen i biblioteket sqlalchemy (Base = declarative\_base()). Dette er et eksempel på anvendelse af polymorphisme.

Indtil videre har vi kun defineret klassen Container i danskcargo\_sql.py. Senere vil vi tilføje flere dataklasser (Aircraft, Transport). sqlalchemy vil overtage opgaven med at tildele dataposterne til de korrekte databasetabeller for os i baggrunden.

Tilføj metoderne update og delete til importen af sqlalchemy:

from sqlalchemy import create\_engine, select, update, delete

Sæt koden for funktionen create\_record() under funktionen get\_record():

def **create\_record**(record): # https://docs.sqlalchemy.org/en/14/tutorial/orm\_data\_manipulation.html#orm-enabled-update-statements

# create a record in the database

with Session(engine) as session:

record.id = None

session.add(record)

session.commit() # makes changes permanent in database

Definer en region “container” under den. Vi indsætter de tre andre funktioner i dette kapitel i denne region.

# region container

# endregion container

Funktionen update\_container() søger i databasen efter en post med et givet ID og opdaterer denne post med værdierne fra parameteren container.

def **update\_container**(container): # https://docs.sqlalchemy.org/en/14/tutorial/orm\_data\_manipulation.html#orm-enabled-update-statements

# update a record in the container table

with Session(engine) as session:

session.execute(update(Container).where(Container.id == container.id).values(weight=container.weight, destination=container.destination))

session.commit() # makes changes permanent in database

Funktionen delete\_hard\_container() søger i databasen efter en post med et givet ID og sletter denne post.

def **delete\_hard\_container**(container):

# delete a record in the container table

with Session(engine) as session:

session.execute(delete(Container).where(Container.id == container.id))

session.commit() # makes changes permanent in database

Ofte er det umuligt, eller det giver ikke mening at slette en post i databasen. Hard delete betyder, at posten fjernes fuldstændigt fra databasen. Men hvis f.eks. postens primærnøgle bruges som en fremmednøgle i en anden tabel i databasen, forhindrer databasen sletningen, så den forbliver konsistent.

Vi vil senere i denne opgave henvise til containerposter i transporttabellen. Så er det ikke længere muligt at slette en containerpost, hvis denne post anvendes i transporttabellen. Derfor sletter vi ikke containerposter hårdt, men blødt. En måde at gøre dette på er at indstille en af beholderens attributter til en "umulig" værdi. Metoden valid() i klassen Container kontrollerer bl.a., om en attribut har denne umulige værdi og så bliver posten ikke vist i GUIen.

def **delete\_soft\_container**(container):

# soft delete a record in the container table by setting its weight to -1 (see also method "valid" in the container class)

with Session(engine) as session:

session.execute(update(Container).where(Container.id == container.id).values(weight=-1, destination=container.destination))

session.commit() # makes changes permanent in database

Nu har du nået den anden milepæl. Du kan sammenligne din løsning med filerne i mappen \milestones\danskcargo\20\_CUD\_in\_sql.

# Milepæl 3: Opret, opdatering, sletning i GUI'en

I det foregående kapitel tilføjede du funktioner til oprettelse, opdatering og sletning af poster i databasen.

I dette kapitel udvider vi GUI'en til at kalde disse funktioner fra GUI'en. Alle ændringerne i dette kapitel skal indarbejdes i filen danskcargo\_gui.py.

Alle tre funktioner følger det samme mønster. Den første linje konverterer en tupel fra treeviewen til et objekt af typen Container. I den anden linje kaldes en af de databasefunktioner, som vi skrev i sidste kapitel. I den tredje linje tømmes indtastningsfelterne (entries). I den sidste linje opdateres container-treeviewen.

Tilføj følgende tre funktioner til regionen “container functions” under funktionen edit\_container():

def **create\_container**(tree, record): # add new tuple to database

container = dcd.Container.convert\_from\_tuple(record) # Convert tuple to Container

dcsql.create\_record(container) # Update database

clear\_container\_entries() # Clear entry boxes

refresh\_treeview(tree, dcd.Container) # Refresh treeview table

def **update\_container**(tree, record): # update tuple in database

container = dcd.Container.convert\_from\_tuple(record) # Convert tuple to Container

dcsql.update\_container(container) # Update database

clear\_container\_entries() # Clear entry boxes

refresh\_treeview(tree, dcd.Container) # Refresh treeview table

def **delete\_container**(tree, record): # delete tuple in database

container = dcd.Container.convert\_from\_tuple(record) # Convert tuple to Container

dcsql.delete\_soft\_container(container) # Update database

clear\_container\_entries() # Clear entry boxes

refresh\_treeview(tree, dcd.Container) # Refresh treeview table

Nu skal knapperne i den grafiske brugergrænseflade forbindes med de funktioner, vi lige har skrevet. Til dette formål tilføjer vi en parameter "command" til definitionerne af knapperne, som indeholder funktionsnavnet. Igen bruger vi tricket med lambda-funktionen. (Se afsnittet "Sammenkobling af widgets med funktioner" i kapitel Milepæl 1.)

button\_create\_container = tk.Button(button\_frame\_container, text=*"Create"*, command=lambda: create\_container(tree\_container, read\_container\_entries()))

button\_update\_container = tk.Button(button\_frame\_container, text=*"Update"*, command=lambda: update\_container(tree\_container, read\_container\_entries()))

button\_delete\_container = tk.Button(button\_frame\_container, text=*"Delete"*, command=lambda: delete\_container(tree\_container, read\_container\_entries()))

Kør programmet. Resultatet bør se nogenlunde sådan ud:



Klik på nogle linjer i treeviewen, rediger værdierne i posterne, og klik derefter på Create eller Update. Test også knappen Delete.

Nu indeholder programmet alt, hvad vi har brug for til Container-klassen. Vi kan læse, oprette, opdatere og slette containere.

Nu har du nået den tredje milepæl. Du kan sammenligne din løsning med filerne i mappen \milestones\danskcargo\30\_CUD\_in\_gui.

# Milepæl 4: Aircraft og Transport, funktionslag

Indtil dette punkt i denne opgave har du kun skullet kopiere den givne kode ind i dine Python-scripts. Nu er det på tide, at du selv skriver noget kode. Din opgave er at kopiere koden for klassen Container to gange og derefter tilpasse kopien til henholdsvis klassen Aircraft og klassen Transport.

Bare rolig, de enkelte mellemliggende trin er beskrevet i detaljer i dette kapitel. Desuden har du altid mulighed for at slå op i koden for milepæl 4, hvis du er i tvivl. Men prøv venligst at løse opgaven selv uden at kigge i milepælen. Det er meget vigtigt, at du selv kan foretage de justeringer, der beskrives i dette kapitel, på egen hånd.

Vi tager et af vores fire Python-scripts ad gangen:

## danskcargo\_data.py

Du skal bruge nogle yderligere objekter fra andre biblioteker. Derfor skal importblokken nu se således ud:

from sqlalchemy.orm import declarative\_base

from sqlalchemy import Column, ForeignKey

from sqlalchemy import String, Integer, Date

from dateutil import parser

from tkinter import messagebox

Dupliker hele koden for klassen Container to gange. Benævn den ene klasse Aircraft og den anden klasse Transport.

### Aircraft

Tilpas nu koden for klassen Aircraft. Læs opgavebeskrivelsen for at finde ud af, hvilke attributter Aircraft skal have.

Giv SQL-tabellen navnet aircraft. Du kan navngive den primære nøgle id, som du gjorde for klassen Container. De to andre attributter skal hedde max\_cargo\_weight og registration. Førstnævnte er af typen Integer og sidstnævnte af typen String. Integer og String er ikke originale Python-datatyper, men foruddefinerede klasser i sqlalchemy, som indeholder databasefunktionalitet.

Tilpas koden for metoderne \_\_repr\_\_(), convert\_to\_tuple(), valid() og convert\_from\_tuple() til de korrekte attributnavne.

### Transport

Tilpas nu koden for Transport-klassen. Læs opgavebeskrivelsen for at finde ud af, hvilke attributter Transport skal have.

Giv SQL-tabellen navnet transport. Ligesom med klassen Container skal du bruge et primærnøgle id. De andre attributter skal hedde date, container\_id og aircraft\_id. Førstnævnte har typen Date, de to andre har typen Integer.

container\_id og aircraft\_id bruges begge som fremmednøgler i tabellen Transport. Derfor modtager Column() to ekstra parametre, når disse attributter defineres:

ForeignKey(*"tablename\_name.attribut\_name"*) og nullable=False)

Den første parameter (ForeignKey()) angiver, at denne attribut bruges her som en fremmed nøgle. ForeignKey() modtager selv en string som argument. Denne string angiver, hvor den fremmede nøgle skal findes, dvs. hvor den bruges som primærnøgle. Den første del af stringen er tabellens navn, og den anden del af stringen er attribut- eller kolonnenavnet i denne tabel. De to dele af teksten er adskilt af et punktum.

Den anden parameter angiver, at denne attribut altid skal indeholde en værdi. Der vil således aldrig være en post i tabellen Transport, hvor en af attributterne container\_id og aircraft\_id ikke er udfyldt.

En dato forekommer i transportklassen. Fordi der er så mange forskellige formater for en dato, kan det være meget tidskrævende at skrive en fleksibel og stabil funktion, der konverterer en tekst til en dato. Du kan simpelthen bruge parse-funktionen fra Parser-biblioteket i convert\_from\_tuple()-metoden i stedet. Din kode til konvertering af indtastningsteksten til en dato vil så se nogenlunde sådan ud:

date = parser.parse(tuple\_[1])

Hvis du er i tvivl om løsningen, kan du naturligvis kopiere den korrekte kode fra den næste milepæl. Men gør dig selv en tjeneste og prøv først at løse opgaven på egen hånd. Det er meget vigtigt, at du lærer dette.

Hvis du sammenligner din kode med koden i den næste milepæl, vil du bemærke, at der i milepælen i metoden convert\_from\_tuple er foretaget et par ekstra tjekks. Det gør programmet mere stabilt og opfanger nogle mulige fejl. Det er dog ikke absolut nødvendigt, at du inddrager disse kontroller i din kode.

## danskcargo\_func.py

Nu kommer der endelig noget kode ind på det såkaldte funktionelle lag. Dette lag indeholder funktioner, der er baseret på indholdet af den konkrete ansøgning.

Vi bruger nu det funktionelle lag til to kontroller fra den oprindelige eksamensopgave (afsnit 2.3.1 Funktionelle kontroller).

Tjek 1: Kan der bookes en anden container på flyet uden at overskride dets maksimale transportkapacitet?

Tjek 2: Passer et flys destination på en given dag med en anden containers destination?

Start med importblokken:

from sqlalchemy.orm import Session

from sqlalchemy import select

from sqlalchemy import extract

import danskcargo\_data as dcd

import danskcargo\_sql as dcsql

Funktionen booked\_cargo() sender en forespørgsel til transporterne i databasen. Dette er begrænset til et bestemt fly på en bestemt dato. Derefter gennemgår en for-loop alle de fundne databaseposter og lægger vægten af containerne sammen.

def **booked\_cargo**(aircraft, date\_):

# returns the already booked cargo on an aircraft at a certain date

with Session(dcsql.engine) as session:

records = session.scalars(select(dcd.Transport).where(dcd.Transport.aircraft\_id == aircraft.id).where(extract(*'day'*, dcd.Transport.date) == date\_.day).where(extract(*'month'*, dcd.Transport.date) == date\_.month).where(extract(*'year'*, dcd.Transport.date) == date\_.year))

weight = 0

for record in records:

weight += dcsql.get\_record(dcd.Container, record.container\_id).weight

return weight

Funktionen capacity\_available() bruger funktionen booked\_cargo() og sammenligner vægten af de containere, der allerede er booket på et bestemt fly på en bestemt dato plus vægten af en ny container, med den samlede vægt, der er tilladt på flyet.

def **capacity\_available**(aircraft, date\_, new\_container):

# do the already booked cargo plus the new container weigh less than the aircraft's maximum cargo weight?

booked = booked\_cargo(aircraft, date\_)

# print(f'{aircraft.max\_cargo\_weight=} {booked=} {new\_container.weight=}')

return aircraft.max\_cargo\_weight >= booked + new\_container.weight

Funktionen find\_destination() foretager en forespørgsel til transporterne i databasen. Dette er begrænset til et bestemt fly på en bestemt dato. Derefter bestemmes containernes destination og dermed flyets destination på denne dato ud fra de fundne databaseposter.

def **find\_destination**(aircraft, date\_):

# return an aircraft's destination at a certain date in the transport table

with Session(dcsql.engine) as session:

records = session.scalars(select(dcd.Transport).where(dcd.Transport.aircraft\_id == aircraft.id).where(extract(*'day'*, dcd.Transport.date) == date\_.day).where(extract(*'month'*, dcd.Transport.date) == date\_.month).where(extract(*'year'*, dcd.Transport.date) == date\_.year))

for record in records:

return dcsql.get\_record(dcd.Container, record.container\_id).destination

return None

Funktionen max\_one\_destination() anvender funktionen find\_destination() og sammenligner destinationen for et givet fly på en given dato med destinationen for en ny container. Resultatet er sandt, hvis destinationerne er identiske.

def **max\_one\_destination**(aircraft, date\_, new\_container):

# is the aircraft's destination at a certain date identical to the new container's destination?

destination = find\_destination(aircraft, date\_)

return destination is None or destination == new\_container.destination # returns also True if aircraft had no destination yet

Man kunne argumentere for, at booked\_cargo() og find\_destination() burde være i SQL-laget, fordi de interagerer med databasen. Men da de blev skrevet specifikt til henholdsvis capacity\_available() og max\_one\_destination() og ikke kaldes af andre funktioner, er det også et godt valg at lade dem blive på funktionslaget.

## danskcargo\_sql.py

I importblokken tilføjes en funktion til at oprette en aktuel dato ud fra heltal, og de nye klasser Aircraft og Transport tilføjes til import fra danskcargo\_data.py. Så ser importblokken ud som følger:

from sqlalchemy.orm import Session

from sqlalchemy import create\_engine, select, update, delete

from datetime import date

from danskcargo\_data import Container, Aircraft, Transport, Base

I funktionen create\_test\_data() tilføjes et par poster for Aircraft og Transport, og rækker med Container-testdata kommenteres ud.

def **create\_test\_data**(): # Optional. Used to test data base functions before gui is ready.

with Session(engine) as session:

new\_items = []

# new\_items.append(Container(weight=1200, destination="Oslo"))

# new\_items.append(Container(weight=700, destination="Helsinki"))

# new\_items.append(Container(weight=1800, destination="Helsinki"))

# new\_items.append(Container(weight=1000, destination="Helsinki"))

new\_items.append(Aircraft(max\_cargo\_weight=2000, registration=*"OY-CBS"*))

new\_items.append(Aircraft(max\_cargo\_weight=3000, registration=*"OY-THR"*))

a\_date = date(day=10, month=12, year=2022)

new\_items.append(Transport(date=a\_date, container\_id=2, aircraft\_id=1))

session.add\_all(new\_items)

session.commit()

Kør nu danskcargo\_sql.py én gang, mens linjen create\_test\_data() ikke er kommenteret ud i hovedprogrammet. Dette vil fylde databasen med nogle testdata for fly og transport. Udkommenter denne linje bagefter.

Kopier nu hele regionen “container” to gange. Denne region indeholder de tre funktioner update\_container(), delete\_hard\_container() og delete\_soft\_container(). Tilpas den første kopi til Aircraft og den anden til Transport. For Transport kan du undlade at bruge funktionen delete\_soft. Vi har ikke brug for det, fordi ingen andre tabeller afhænger af transporterne, og derfor kan vi til enhver tid slette dem hårdt.

## danskcargo\_gui.py

I importblokken skal du tilføje messagebox fra tkinter (til at udsende fejlmeddelelser) og også importere vores funktionslag. Så ser importblokken således ud:

### Import

import tkinter as tk

from tkinter import ttk

from tkinter import messagebox

import danskcargo\_data as dcd

import danskcargo\_sql as dcsql

import danskcargo\_func as dcf

### Klassespecifikke funktioner

Kopier nu hele regionen "container functions" to gange. Tilpas den første kopi til Aircraft og den anden til Transport.

Funktionerne Create og Update er noget mere komplekse for klassen Transport end for Aircraft og Container, fordi de to kontroller, som vi tidligere havde skrevet i funktionslaget, også udføres her. Her er derfor koden for create\_transport():

def **create\_transport**(tree, record): # add new tuple to database

transport = dcd.Transport.convert\_from\_tuple(record) # Convert tuple to Transport

capacity\_ok = dcf.capacity\_available(dcsql.get\_record(dcd.Aircraft, transport.aircraft\_id), transport.date, dcsql.get\_record(dcd.Container, transport.container\_id))

destination\_ok = dcf.max\_one\_destination(dcsql.get\_record(dcd.Aircraft, transport.aircraft\_id), transport.date, dcsql.get\_record(dcd.Container, transport.container\_id))

if destination\_ok:

if capacity\_ok:

dcsql.create\_record(transport) # Update database

clear\_transport\_entries() # Clear entry boxes

refresh\_treeview(tree, dcd.Transport) # Refresh treeview table

else:

messagebox.showwarning(*""*, *"Not enough capacity on aircraft!"*)

else:

messagebox.showwarning(*""*, *"Aircraft already has another destination!"*)

Kopier koden fra create\_transport() og omdøb kopien til update\_transport(). I kopien behøver du kun at tilpasse denne linje:

dcsql.create\_record(transport) # Update database

Find ud af hvordan.

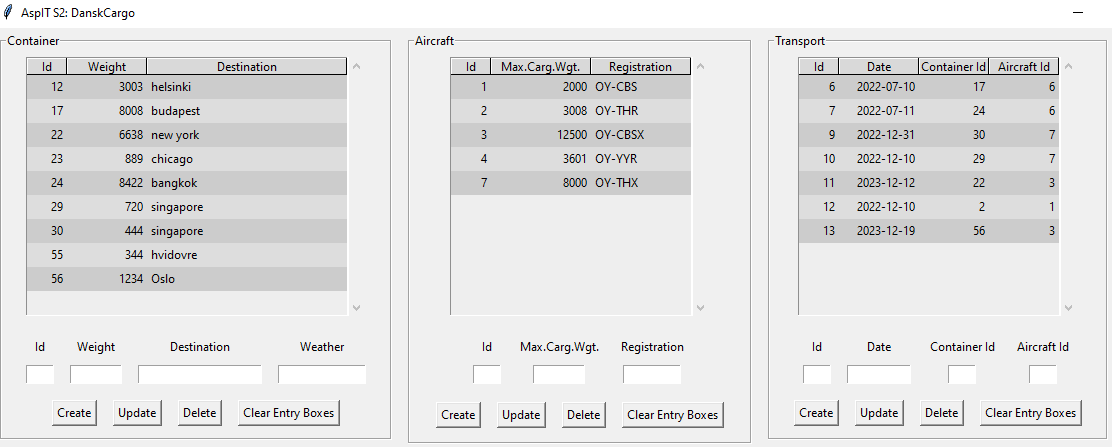
### Widgets

Det kan være nødvendigt at forstørre hovedvinduet i dit program for at få plads til de ekstra GUI-komponenter til fly og transport:

main\_window.geometry(*"1200x500"*) # window size

Kopier nu hele regionen "container widgets" to gange. Tilpas den første kopi til Aircraft og den anden til Transport.

Kør programmet. Resultatet skal se således ud:



Test programmet. Opret, opdater og slet containere, fly og transporter.

Bemærk, at programmet opfanger nogle fejl, men ikke andre. Det ville kræve en betydelig ekstra indsats at forhindre enhver fejlbetjening.

Nu har du nået den fjerde milepæl. Du kan sammenligne din løsning med filerne i mappen \milestones\danskcargo\40\_aircraft\_transport\_funcpy.

# Milepæl 5: API, Unittest

API og Unittests er ikke længere en del af S2.

Nu har du nået den sidste milepæl. Du kan sammenligne din løsning med filerne i mappen \milestones\danskcargo\90\_api+unittest.