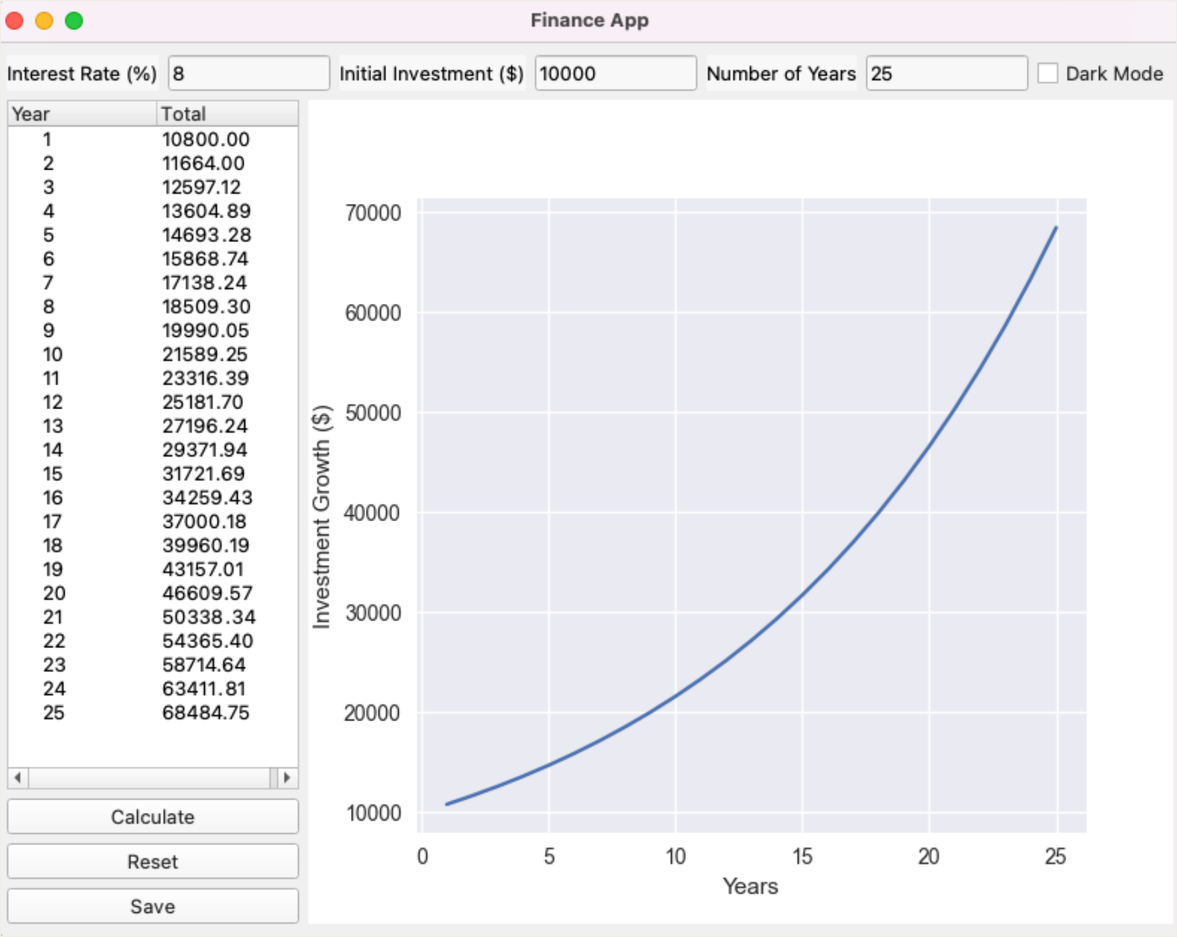
INTEREST RATE APP

Introduction to Data Visualization using PyQt5 and MatplotLib

ZERO TO KNOWING

App Overview

What Widgets do you see?



Let's take a look at the App we will be building:

We can:

-We **enter** a **rate**, **amount** & **years**

-A **chart is created**

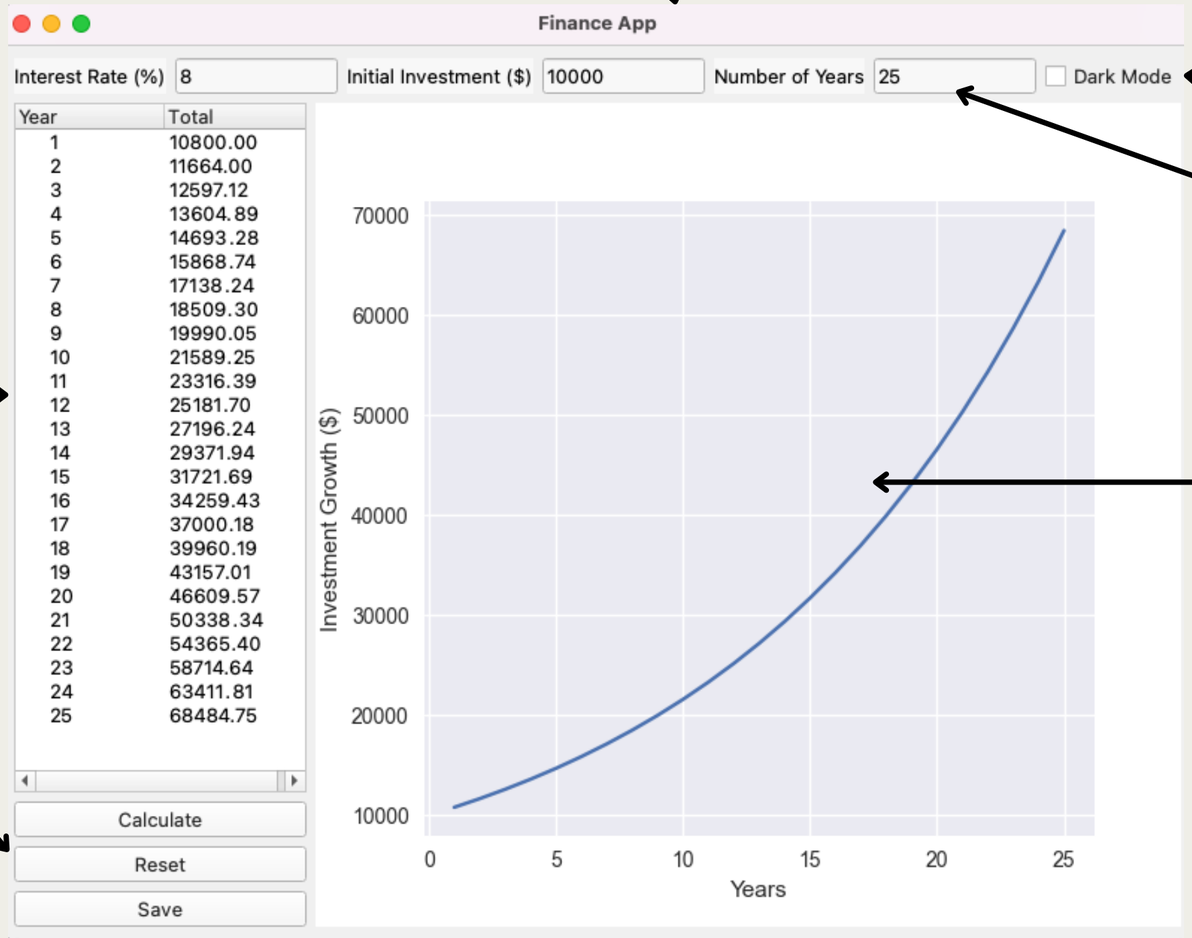
-The **Rate is displayed in a table** too

-**Ability to save** the Chart and table as a .csv file

App Overview

What Widgets do you see?

QWidget



QCheckBox

QLineEdit

QTreeView

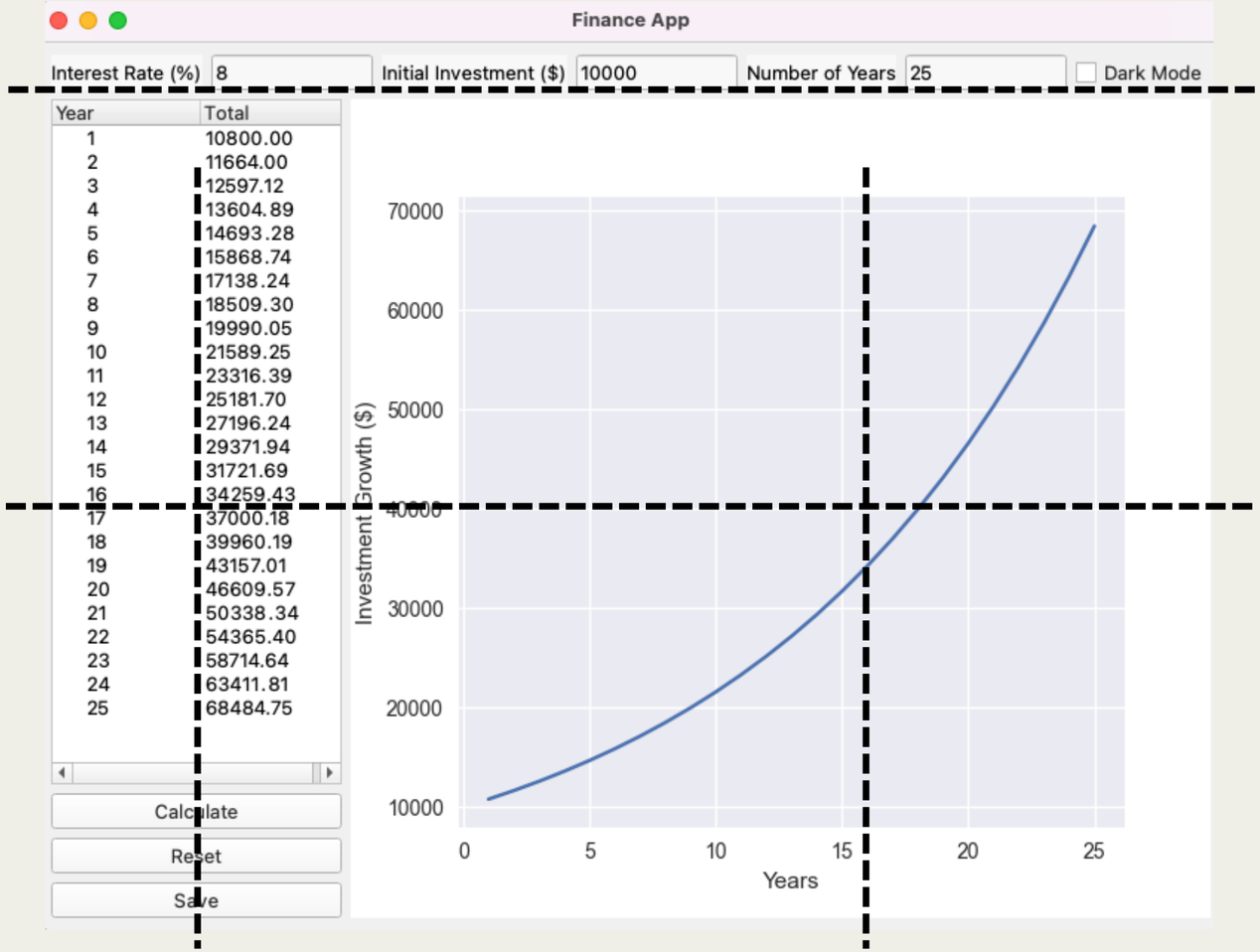
?????

QPushButton

For now let's just use, QLabel

App Design

How can we design this?



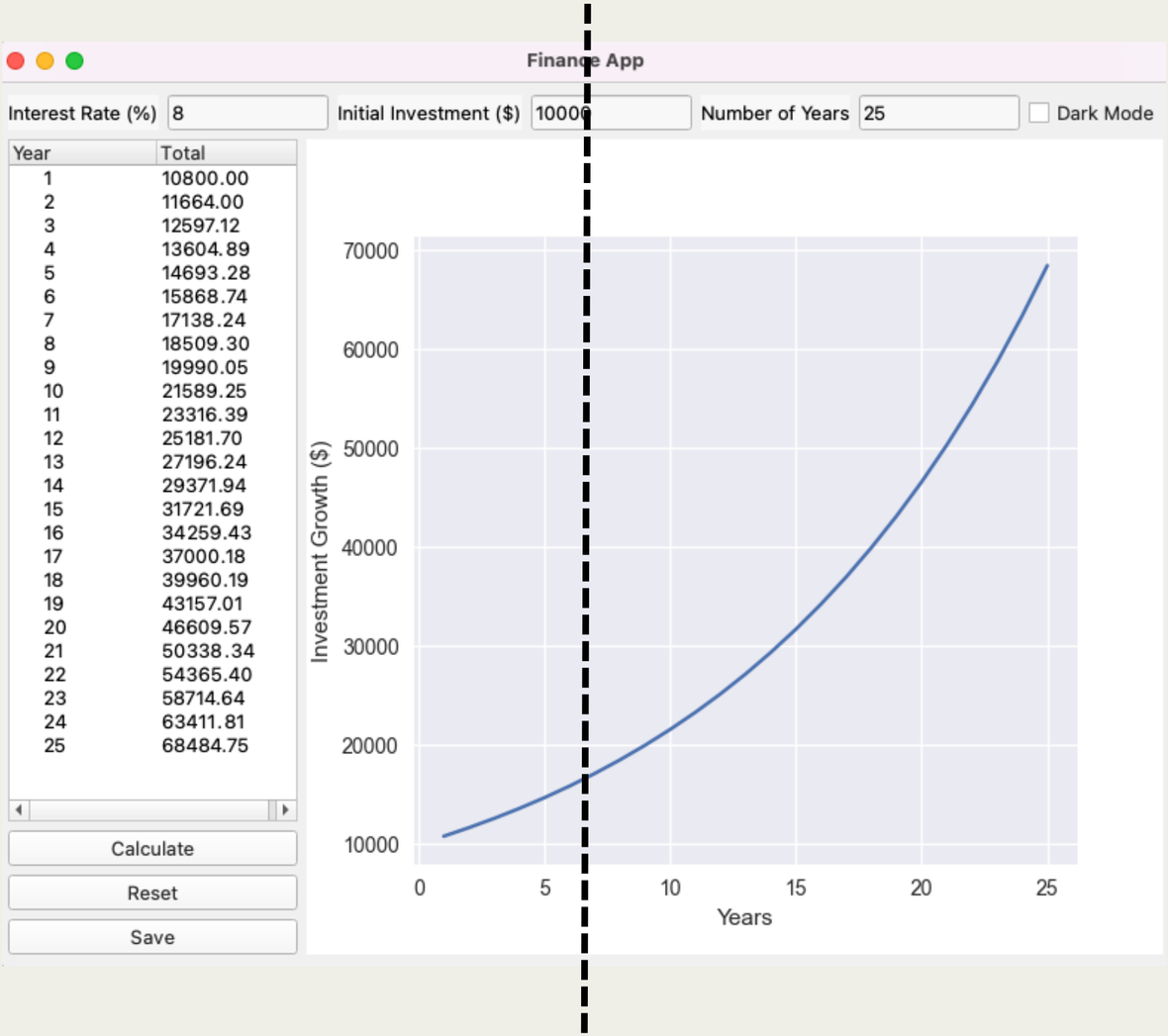
Our Design will have **Two Rows**

Row 2 will have 2 Columns

These Two Rows will be held in a **master_layout column**

App Overview

What Widgets do you see?



master_layout

Everything is held in a final master_layout. This is a **Column**

Initial App Setup

Based on your PyQt experience, Try to set the App up!

Calculate Interest

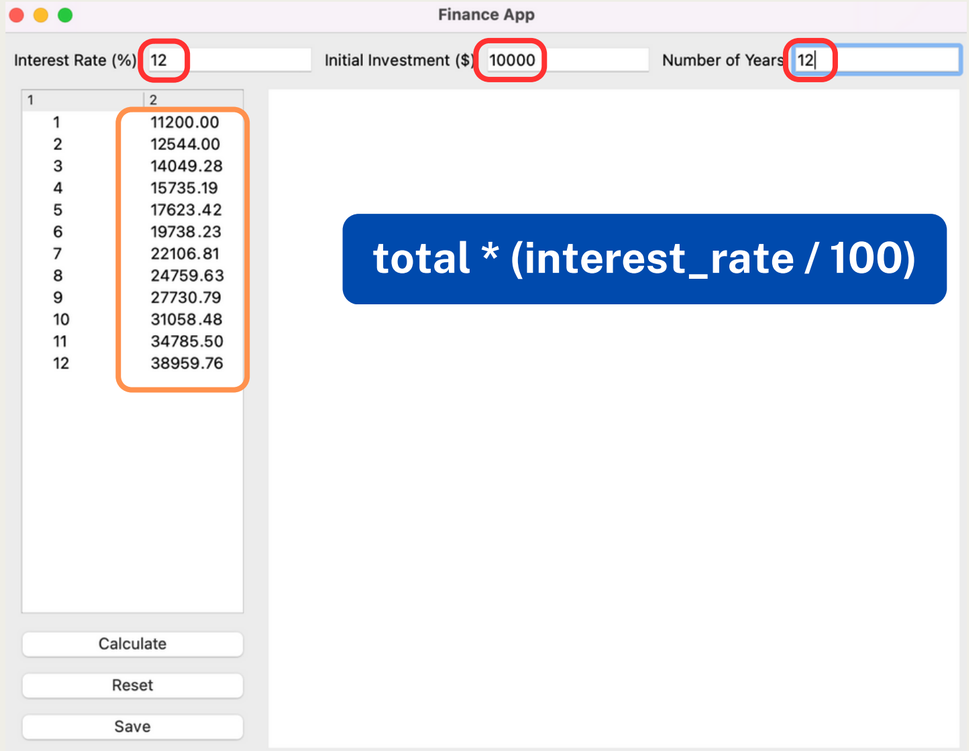
How can we Calculate the Interest and create a chart?

Calculate Interest

ZERO TO KNOWING

Add Interest Calculation to QTreeView

- **Convert all** of our **input fields** to **numbers**
- **Catch** any possible **errors**
- **For** every year, multiply the **total investment** by **interest rate**
  total * (interest_rate / 100)
- Create **TreeView items** with **QStandardItem**
- **Add item_year** and **item_total** to our QTreeView **as a List**
- **Create a Save Button** and add to Layout



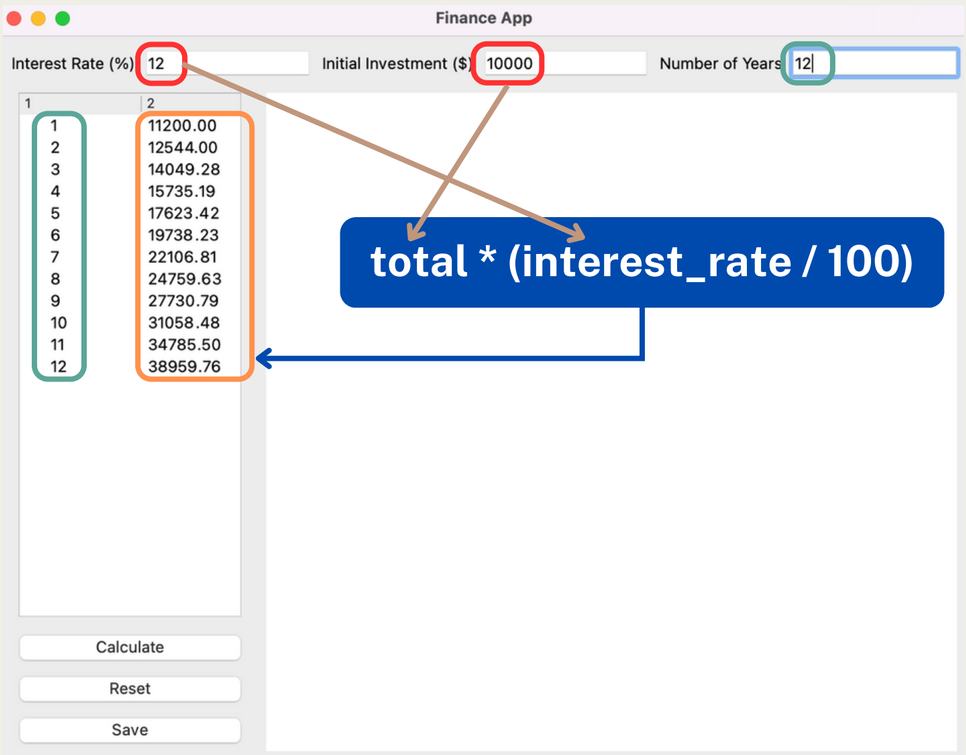
total * (interest_rate / 100)

Calculate Interest

ZERO TO KNOWING

Add Interest Calculation to QTreeView

- **Convert all** of our **input fields** to **numbers**
- **Catch** any possible **errors**
- **For** every year, multiply the **total investment** by **interest rate**
  total * (interest_rate / 100)
- Create **TreeView items** with **QStandardItem**
- **Add item_year** and **item_total** to our QTreeView **as a List**
- **Create a Save Button** and add to Layout



Converting our Input Fields

ZERO TO KNOWING

```
try:
    interest_rate = float(self.rate_input.text())
    initial_investment = float(self.investment_input.text())
    num_years = int(self.years_input.text())

 except ValueError:
    error_message = "Invalid input. Please enter a valid number"
    QMessageBox.warning(self, "Error", error_message)
     return
```

We **try** to take our values we entered from our **Input Fields** and **convert** them to **floats/ints**

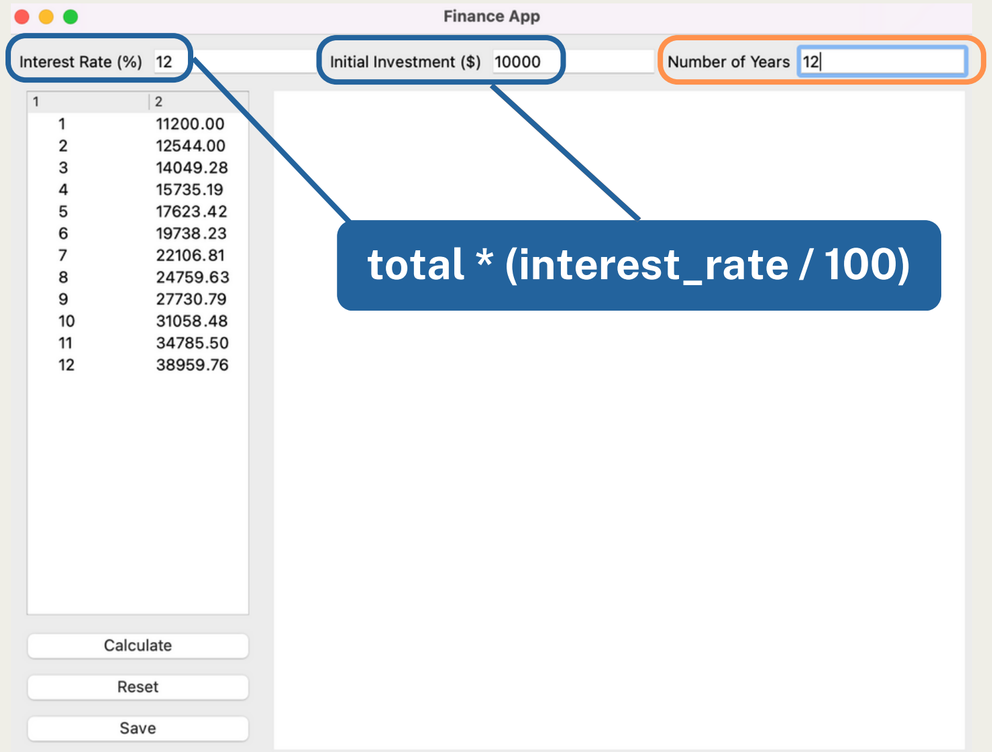
If the **user enters**, something besides a number, we t**hrow a Value Error**. We give them a **Pop-up message**

Calculating Interest

ZERO TO KNOWING

This is the number we collected from our Input Field

```
total = initial_investment
for year in range(1, num_years + 1):
    total += total * (interest_rate / 100)
    item_year = QStandardItem(str(year))
    item_total = QStandardItem("{:.2f}".format(total))
    self.model.appendRow( [item_year, item_total] )
```



***A string format specifier that formats a floating-point number with two decimal places

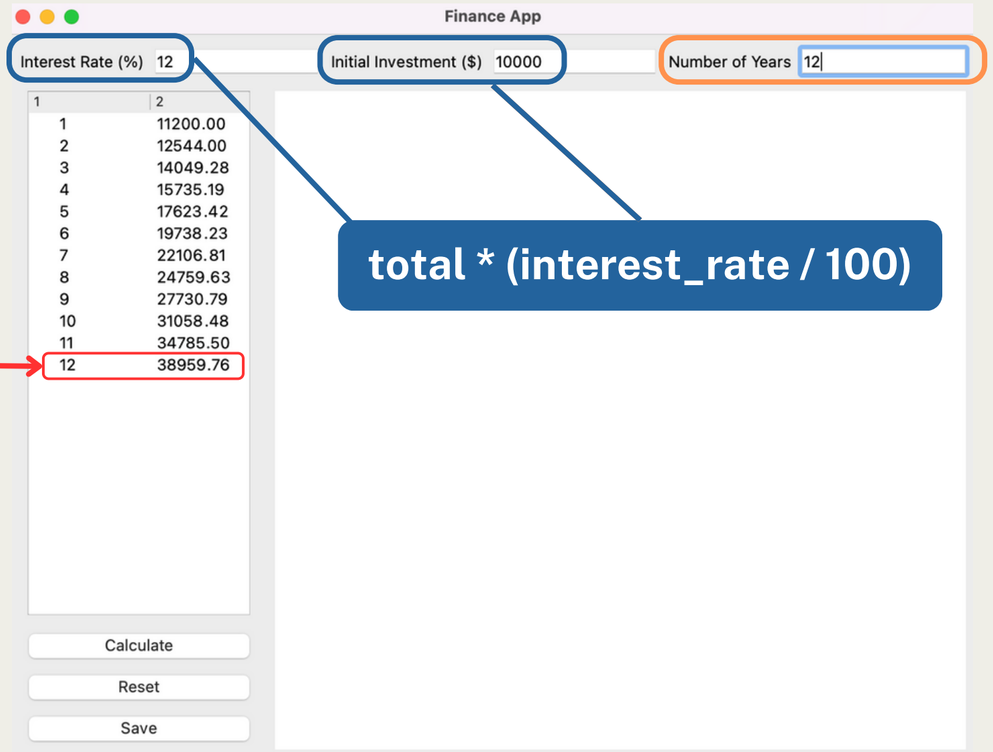
The ".2f" specifies that the number should be formatted with two digits after the decimal point

Calculating Interest

ZERO TO KNOWING

This is the number we collected from our Input Field

```
total = initial_investment
for year in range(1, num_years + 1):
    total += total * (interest_rate / 100)
    item_year = QStandardItem(str(year))
    item_total = QStandardItem("{:.2f}".format(total))
    self.model.appendRow([item_year, item_total])
```



total * (interest_rate / 100)

Bonus: Create a method to reset the app (clear)

ZERO TO KNOWING

Intro to MatplotLib

Getting started with Data Visualization in PyQt

MatplotLib Imports & Classes

ZERO TO KNOWING

```
import matplotlib.pyplot as plt
```

importing matplotlib and giving it the **nickname**, **plt**

```
from matplotlib.backends.backend_qt5agg import FigureCanvasQTagg as FigureCanvas
```

Importing a Class called **FigureCanvasQTagg**

We then **shorten** the name **to FigureCanvas**

FigureCanvas

This acts as a **bridge between MatplotLib and PyQt**

This class allows you to **create a canvas object** that **acts as a container** for Matplotlib figures, **enabling** the integration of **Matplotlib plots with PyQt**

Data Visualization

One of the most powerfuls Python tools, **MatplotLib**
We will use this create **simple plots**, line **graphs** and/or scatter graphs

MatplotLib Methods	What it does
.subplots()	Can **generate** one or more **plots** in a single figure
.plot()	Try to **plot the data** that it is given
.show()	Opens and **displays the plot** once completed
.set_xlabel() / .set_ylabel() / .set_title()	Allows you to **give a Title** to the Chart, X Axis and Y Axis
.figure()	**Create a new figure** or get a reference to an existing figure
.draw()	Used to explicitly **redraw the figure** and update its contents

Create our Plot with Code

ZERO TO KNOWING

```
#Change our current Chart Object and add to Layout

self.figure = plt.figure()
self.canvas = FigureCanvas(self.figure)
 self.row2.addWidget(self.canvas)
```

Changing our property to Create a new figure

Giving our figure to FigureCanvas

This will act as a container to work with PyQt and MatplotLib

Adding our Plot to our Layout

Updating our Plot

Can you reorganize the steps?

```
#Update the Chart w/ our results
self.figure.clear()
 ax = self.figure.subplots()
 years = list(range(1, num_years + 1))
 totals = [initial_investment * (1 + interest_rate / 100) ** year for year in years]
 ax.plot( years, totals )
 ax.set_title("Interest")
 ax.set_xlabel("Year")
 ax.set_ylabel("Total")
 self.canvas.draw()
```

- Generating a List of Years based on our input
- Update our Canvas
- Setting all our Titles
- List Comprehension - Creating a List of Interest over the Years
- Plot the Data
- Creating a new plot

Updating our Plot

We update our Plot within our compound method

```
#Update the Chart w/ our results
self.figure.clear()
 ax = self.figure.subplots()
 years = list(range(1, num_years + 1))
 totals = [initial_investment * (1 + interest_rate / 100) ** year for year in years]
 ax.plot( years, totals )
 ax.set_title("Interest")
 ax.set_xlabel("Year")
 ax.set_ylabel("Total")
 self.canvas.draw()
```

Creating a new plot

Generating a List of Years based on our input

List Comprehension - Creating a List of Interest over the Years

Plot the Data

Setting all our Titles

Update our Canvas

List Comprehension

Breakdown of List Comprehension

```
totals = [ ]

for i  in range(15):
    totals.append(25000 * (1 + 10 / 100) ** i )
```

Length of Years

initial investment

interest rate

Current Year

Can we use our Input Values for this?

List Comprehension

Breakdown of List Comprehension

```
totals = [ ]

for i  in range( 15 ):
    totals.append( 25000 * (1 + 10 / 100) ** i )
```

Length of Years

initial investment

interest rate

Current Year

Can we use our Input Values for this?

```
totals = [ ]

for i  in range( 15 ):
    totals.append( inital_investment * (1 + interest_rate / 100) ** i )
```

List Comprehension

Breakdown of List Comprehension

```
totals = [ ]

for i  in range( 15 ):
    totals.append( inital_investment * (1 + interest_rate / 100) ** i )
```

How can I combine these three lines of code into one?

List Comprehension

Breakdown of List Comprehension

ZERO TO KNOWING

```
totals = [ ]

for i  in range( 15 ):
    totals.append( inital_investment * (1 + interest_rate / 100) ** i )
```

```
totals = [initial_investment * (1 + interest_rate / 100) ** year for year in years]
```

Defining a new list called totals

Calculation for each element in the list. This takes the initial investment and multiplies it by the interest rate, squaring it to the current year

Repeat the number of years in my list years

Adding Save

Adding the Save ability for our Table and Chart

Does this look familiar?

ZERO TO KNOWING

This is our save method from our Image App

How could we create a similar method now?

Remember OS -> Path -> **Join**?
We use this to **link** our working **directory**

```
def saveImage(self):
    path = os.path.join( work_directory, self.save_folder )
    if not(os.path.exists( path ) or os.path.isdir( path )):
        os.mkdir( path )
    fullname = os.path.join(path, self.filename)
    self.image.save( fullname )
```

We can **make** a new **directory**

Our Save Method

ZERO TO KNOWING

We can create a **Save Folder**, a **CSV File** and a picture of our **Chart**

```
dir_path = QFileDialog.getExistingDirectory(self, "Select Directory")
if dir_path:
    # Create a subfolder within the selected directory
    folder_path = os.path.join(dir_path, "Saved")
    os.mkdir(folder_path)

    # Save the results to a CSV file within the subfolder
    file_path = os.path.join(folder_path, "results.csv")
    with open(file_path, "w") as file:
        file.write("Year,Total\n")
        for row in range( self.model.rowCount() ):
            year = self.model.index(row, 0).data()
            total = self.model.index(row, 1).data()
            file.write("{},{}\n".format(year, total))
```

Creating a **"saved" folder** to save the charts and csv files to

Collect the **Data** in the current row

Write the data in **this format** in a **CSV** File

Our Save Method

We can use the handy matplotlib method .**savefig()** this works similar to the .s**ave()** method we have use before

matplotlib method to save an image of the chart/plot

```
plt.savefig(f"Results/chart.png")

# Show a message box to indicate successful save
QMessageBox.information(self, "Save Results", "Results saved successfully in
'{}'".format(folder_path))
else:
    QMessageBox.warning(self, "Save Results", "No directory selected.")
```

A friendly pop-up to show that the save was a success

ZERO TO KNOWING

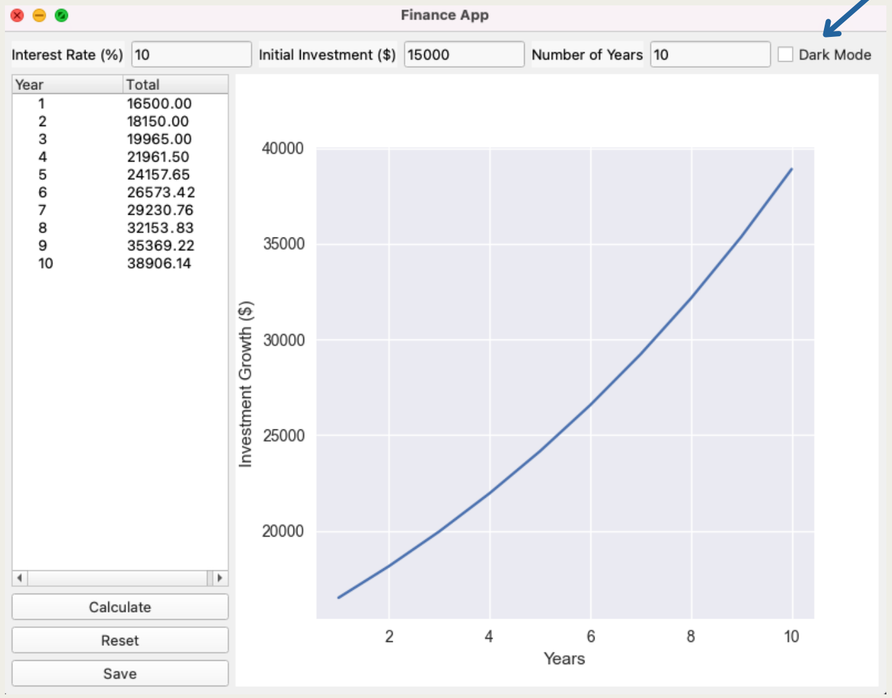
Dark Mode

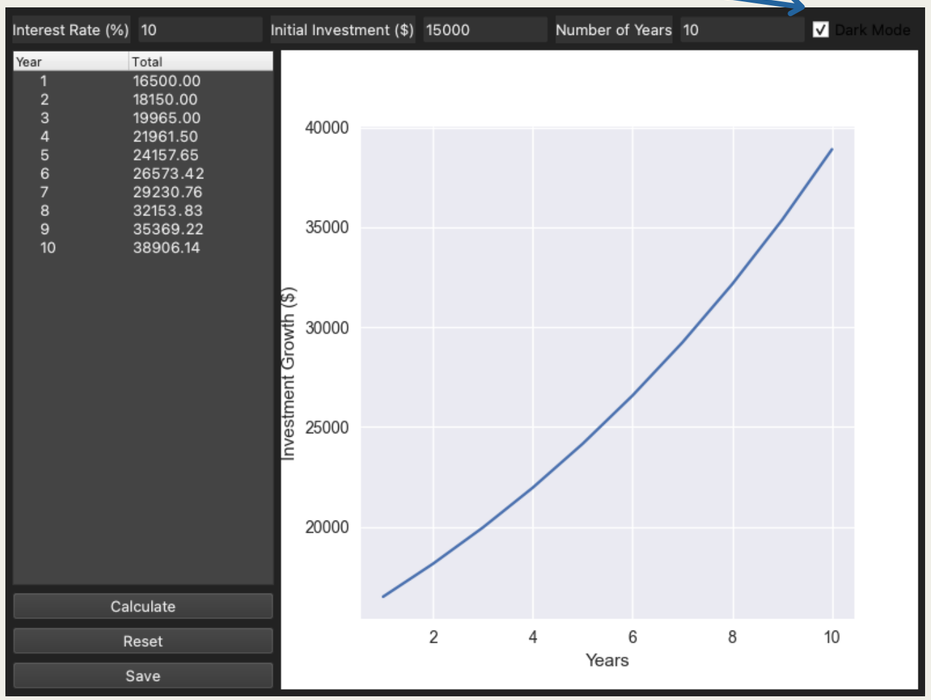
Adding a Dark Mode with Styles to our App

Adding a bonus dark mode

The last thing we could add once our app is working are any design features and styling. Such as Fonts, Colors and Styling

Dark Mode





Adding CSS to a PyQt App

ZERO TO KNOWING

```
def apply_styles(self):

    # Set dark mode styles
    if self.dark_mode_checkbox.isChecked():
        self.setStyleSheet(
            """
            FinanceApp {
                background-color: #222222;
            }
            QLabel, QLineEdit, QPushButton {
                background-color: #333333;
                color: #eeeeee;
            }
            QTreeView {
                background-color: #444444;
                color: #eeeeee;
            }
            """
        )
```

isChecked - checks the **state** of a QComboBox

setStyleSheet - Allows for **CSS** in PyQt

Do we remember any **CSS**?

```
h1 {
    color: #fff;
    font-size: 32px;
    font-family: gothic;
}
```

Adding CSS to a PyQt App

ZERO TO KNOWING

```
def apply_styles(self):

    # Set dark mode styles
    if self.dark_mode_checkbox.isChecked():
        self.setStyleSheet(
            """
            FinanceApp {
                background-color: #222222;
            }
            QLabel, QLineEdit, QPushButton {
                background-color: #333333;
                color: #eeeeee;
            }
            QTreeView {
                background-color: #444444;
                color: #eeeeee;
            }
            """
        )
```

***In **PyQt** CSS is **styled** in a **string**

isChecked - checks the **state** of a QComboBox

setStyleSheet - Allows for **CSS** in PyQt

```
h1 {
    color: #fff;
    font-size: 32px;
    font-family: gothic;
}
```

The element you want to effect

Specific styling x 3

How to Toggle Dark Mode

ZERO TO KNOWING

```
def toggle_dark_mode(self):
    self.apply_styles()
```

Method that simply calls our styling method

Creating a Dark Mode Box

```
self.dark_mode_checkbox = QCheckBox("Dark Mode")
self.dark_mode_checkbox.stateChanged.connect(self.toggle_dark_mode)
row1.addWidget(self.dark_mode_checkbox)
```

An **event** that **triggers** when the **state** of that object is **changed**

(Ex. when the object is clicked)

ZERO TO KNOWING

WRAP UP