

# Advanced PyQt5 tutorial

Jan Bodnar

November 2, 2017

# Contents

|  |           |
|--|-----------|
| <b>About the author</b>                  | <b>iv</b> |
| <b>About the e-book</b>                  | <b>v</b>  |
| <b>1 Date and time</b>                   | <b>1</b>  |
| 1.1 Initializing date and time . . . . . | 1         |
| 1.2 Current date and time . . . . .      | 2         |
| 1.3 UTC time . . . . .                   | 3         |
| 1.4 Comparing dates . . . . .            | 4         |
| 1.5 Weekday . . . . .                    | 4         |
| 1.6 Number of days . . . . .             | 5         |
| 1.7 Day of week, month, year . . . . .   | 5         |
| 1.8 Difference in days . . . . .         | 6         |
| 1.9 Datetime arithmetic . . . . .        | 6         |
| 1.10 Daylight saving time . . . . .      | 7         |
| 1.11 Leap year . . . . .                 | 8         |
| 1.12 Unix epoch . . . . .                | 9         |
| 1.13 Julian day . . . . .                | 10        |
| <b>2 Graphics</b>                        | <b>12</b> |
| 2.1 Lines . . . . .                      | 12        |
| 2.2 Drawing text . . . . .               | 15        |
| 2.3 Grayscale image . . . . .            | 17        |
| 2.4 Donut . . . . .                      | 19        |
| 2.5 Shapes . . . . .                     | 21        |
| 2.6 Transparent rectangles . . . . .     | 23        |
| 2.7 Gradients . . . . .                  | 24        |
| 2.8 Waiting effect . . . . .             | 26        |
| 2.9 Puff effect . . . . .                | 29        |
| 2.10 Hit test . . . . .                  | 31        |
| 2.11 Image Reflection . . . . .          | 34        |
| 2.12 Clipping . . . . .                  | 38        |
| 2.13 Moving star . . . . .               | 41        |
| <b>3 The Graphics View framework</b>     | <b>45</b> |
| 3.1 Simple example . . . . .             | 45        |
| 3.2 Custom graphics item . . . . .       | 47        |
| 3.3 Rotating a rectangle . . . . .       | 49        |

|          |   |            |
|----------|---|------------|
| 3.4      | Item animation . . . . .                            | 52         |
| 3.5      | Collision detection . . . . .                       | 54         |
| 3.6      | Selecting items . . . . .                           | 58         |
| 3.7      | Zooming items . . . . .                             | 61         |
| 3.8      | Grouping items . . . . .                            | 63         |
| 3.9      | Progress meter . . . . .                            | 68         |
| 3.10     | Rotating arrow example . . . . .                    | 72         |
| 3.11     | Aliens . . . . .                                    | 78         |
| <b>4</b> | <b>Layout Management</b>                            | <b>88</b>  |
| 4.1      | Absolute positioning . . . . .                      | 88         |
| 4.2      | Box layout . . . . .                                | 90         |
| 4.3      | Size policy . . . . .                               | 91         |
| 4.4      | Size hint . . . . .                                 | 92         |
| 4.5      | Minimum and maximum size . . . . .                  | 93         |
| 4.6      | Nesting layout managers . . . . .                   | 94         |
| 4.7      | Stretch factor . . . . .                            | 96         |
| 4.8      | Alignment . . . . .                                 | 98         |
| 4.9      | Buttons example . . . . .                           | 101        |
| 4.10     | Windows example . . . . .                           | 102        |
| 4.11     | FindReplace example . . . . .                       | 104        |
| 4.12     | QGridLayout manager . . . . .                       | 107        |
| 4.12.1   | Simple example . . . . .                            | 107        |
| 4.12.2   | Spacing . . . . .                                   | 108        |
| 4.12.3   | Spanning . . . . .                                  | 110        |
| 4.12.4   | Alignments . . . . .                                | 111        |
| 4.12.5   | New folder example . . . . .                        | 112        |
| 4.12.6   | Windows example . . . . .                           | 114        |
| <b>5</b> | <b>Custom widgets</b>                               | <b>116</b> |
| 5.1      | Led widget . . . . .                                | 116        |
| 5.2      | CPU widget . . . . .                                | 120        |
| 5.3      | Thermometer widget . . . . .                        | 124        |
| <b>6</b> | <b>Model and View widgets</b>                       | <b>131</b> |
| 6.1      | Basic examples . . . . .                            | 131        |
| 6.1.1    | QTreeView basic example . . . . .                   | 131        |
| 6.1.2    | QListView basic example . . . . .                   | 132        |
| 6.1.3    | QTableView basic example . . . . .                  | 134        |
| 6.2      | Sorting . . . . .                                   | 136        |
| 6.2.1    | Built-in sorting . . . . .                          | 136        |
| 6.2.2    | Sorting with proxy models . . . . .                 | 139        |
| 6.3      | QModelIndex . . . . .                               | 141        |
| 6.4      | QItemSelectionModel . . . . .                       | 144        |
| 6.5      | Subclassing models . . . . .                        | 147        |
| 6.6      | Filtering data with QSortFilterProxyModel . . . . . | 152        |
| 6.7      | Delegates . . . . .                                 | 155        |

|           |  |            |
|-----------|--|------------|
| <b>7</b>  | <b>QtSql module</b>                          | <b>163</b> |
| 7.1       | SQLite database . . . . .                    | 163        |
| 7.2       | QSqlQuery . . . . .                          | 164        |
| 7.2.1     | Database version . . . . .                   | 164        |
| 7.2.2     | Creating a table . . . . .                   | 166        |
| 7.2.3     | Selecting rows . . . . .                     | 167        |
| 7.2.4     | Prepared statements . . . . .                | 169        |
| 7.3       | Metadata . . . . .                           | 170        |
| 7.3.1     | Driver features . . . . .                    | 170        |
| 7.3.2     | Column names . . . . .                       | 171        |
| 7.3.3     | Affected rows . . . . .                      | 173        |
| 7.4       | Transactions . . . . .                       | 174        |
| 7.5       | Model and View . . . . .                     | 176        |
| 7.5.1     | Read-only model . . . . .                    | 177        |
| 7.5.2     | Editable model . . . . .                     | 180        |
| 7.5.3     | Relational model . . . . .                   | 182        |
| <b>8</b>  | <b>Networking</b>                            | <b>186</b> |
| 8.1       | Network interfaces . . . . .                 | 186        |
| 8.2       | IP address, host name . . . . .              | 188        |
| 8.3       | QNetworkAccessManager . . . . .              | 191        |
| 8.3.1     | HTTP GET request . . . . .                   | 191        |
| 8.3.2     | HTTP POST request . . . . .                  | 193        |
| 8.3.3     | HTTP HEAD request . . . . .                  | 194        |
| 8.3.4     | Authentication . . . . .                     | 196        |
| 8.3.5     | Fetching a favicon . . . . .                 | 198        |
| 8.4       | Blocking and non-blocking requests . . . . . | 199        |
| 8.5       | Echo service . . . . .                       | 204        |
| 8.6       | Web server . . . . .                         | 208        |
| 8.7       | Connecting to an SMTP server . . . . .       | 212        |
| 8.8       | Weather conditions . . . . .                 | 214        |
| <b>9</b>  | <b>Games</b>                                 | <b>218</b> |
| 9.1       | Snake . . . . .                              | 218        |
| 9.2       | Sokoban . . . . .                            | 225        |
| 9.3       | Minesweeper . . . . .                        | 238        |
| <b>10</b> | <b>References</b>                            | <b>249</b> |

# About the author

My name is Jan Bodnar. I am Hungarian and I come from Slovakia. Currently I live in Bratislava. Apart from working with computers I read a lot. Mostly belles-letters and history. I study several foreign languages. I dance, do Tai chi, work in the garden, and often take tours.

I run the [zetcode.com](http://zetcode.com) site. My other e-books are The Swing layout management, MySQL Java programming, Advanced wxPython, Advanced Java Swing, Tkinter programming, Introduction to Windows API programming, and SQLite Python programming. I hope this e-book will help you become a better programmer.



Thank you for buying this e-book.

# About the e-book

This is Advanced PyQt5 e-book. This e-book covers several parts of the PyQt5 library in a great detail.

This e-book covers the following:

- Date and time
- Graphics
- Graphics View framework
- Layout Management
- Custom widgets
- Model and View widgets
- QSql module
- Networking
- Games

The first chapter covers date and time. The examples demonstrate date and time arithmetic, show how to work with universal and local time, epochs, or daylight saving time. In the Graphics chapter, we do some basic and more advanced painting. The Graphics View framework chapter covers Graphics View framework, which is a high level framework to create graphics quickly and efficiently.

In the Layout Management chapter, we show how to use built-in layout management classes to lay out our widgets on the window. In the Custom widgets part, we show how to create custom widgets. We create a Led widget, a CPU widget and a Thermometer widget. In the Model and View widgets chapter, we cover important Model and View widgets. We learn how to work with `QTableView`, `QTreeView`, and `QListView`.

In the QSql module chapter, we cover the basics of the QSql, which is used for database programming. Network programming is covered in the networking chapter. The examples show how to download an icon from a web page, create a simple web server, or determine weather conditions. One of the best ways to study programming is to create simple computer games. In this e-book, we create three 2D games called Snake, Sokoban, and Minesweeper.

In all chapters, you will find lots of practical examples. Many of them cannot be found anywhere else.

The E-book was written in August 2010. In November 2013, the first revision was finished. In November 2017, a second revision was finished.

# Chapter 1

## Date and time

PyQt5 has `QDate`, `QDateTime`, `QTime` classes to work with date and time. The `QDate` is a class for working with a calendar date in the Gregorian calendar. It has methods for determining the date, comparing, or manipulating dates. The `QTime` class works with a clock time. It provides methods for comparing time, determining the time and various other time manipulating methods. The `QDateTime` is a class that combines both `QDate` and `QTime` objects into one object.

### 1.1 Initializing date and time

PyQt5 has various constructors to initialize date and time objects.

Listing 1.1: Initializing date and time

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, QTime, QDateTime, Qt

d1 = QDate(2017, 12, 9)
t1 = QTime(18, 50, 59)

dt1 = QDateTime(d1, t1, Qt.LocalTime)

print("Datetime: {}".format(dt1.toString()))
print("Date: {}".format(d1.toString()))
print("Time: {}".format(t1.toString()))
```

---

In the example we initialize date, time, and datetime objects.

```
from PyQt5.QtCore import QDate, QTime, QDateTime, Qt
```

We import the necessary classes.

```
d1 = QDate(2017, 12, 9)
t1 = QTime(18, 50, 59)

dt1 = QDateTime(d1, t1, Qt.LocalTime)
```

Here we create instances of `QDate`, `QTime`, and `QDateTime`.

```
print("Datetime: {}".format(dt1.toString()))
```



```
print("Date: {0}".format(d1.toString()))
print("Time: {0}".format(t1.toString()))
```

We print the objects to the console with `toString()`.

```
$ ./initializing.py
Datetime: Sat Dec 9 18:50:59 2017
Date: Sat Dec 9 2017
Time: 18:50:59
```

This is the output.

## 1.2 Current date and time

Another common way of initializing date and time objects is getting the current date and time.

---

Listing 1.2: Current date and time

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, QTime, QDateTime, Qt

now = QDate.currentDate()

print(now.toString(Qt.ISODate))
print(now.toString(Qt.DefaultLocaleLongDate))

datetime = QDateTime.currentDateTime()

print(datetime.toString())

time = QTime.currentTime()

print(time.toString(Qt.DefaultLocaleLongDate))
```

---

The example prints the current date, date and time, and time in various formats.

```
now = QDate.currentDate()
```

The `currentDate()` method returns the current date.

```
print(now.toString(Qt.ISODate))
print(now.toString(Qt.DefaultLocaleLongDate))
```

The date is printed in two different formats by passing the values `Qt.ISODate` and `Qt.DefaultLocaleLongDate` to the `toString()` method.

```
datetime = QDateTime.currentDateTime()
```

The `currentDateTime()` returns the current date and time.

```
time = QTime.currentTime()
```

Finally, the `currentTime()` method returns the current time.

```
$ ./current_date_time.py
2017-09-11
Monday, September 11, 2017
Mon Sep 11 12:37:45 2017
12:37:45 PM CEST
```

This is the output.

## 1.3 UTC time

Our planet is a sphere; it revolves round its axis. The Earth rotates towards the east, so the Sun rises at different times in different locations. The Earth rotates once in about 24 hours. Therefore, the world was divided into 24 time zones. In each time zone, there is a different local time. This local time is often further modified by the daylight saving.

There is a pragmatic need for one global time. One global time helps to avoid confusion about time zones and daylight saving time. The UTC (Universal Coordinated time) was chosen to be the primary time standard. UTC is used in aviation, weather forecasts, flight plans, air traffic control clearances, and maps. Unlike local time, UTC does not change with a change of seasons.

Listing 1.3: Universal time

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDateTime, Qt

now = QDateTime.currentDateTime()

print("Local datetime: ", now.toString(Qt.ISODate))
print("Universal datetime: ", now.toUTC().toString(Qt.ISODate))

print("The offset from UTC is: {0} seconds".format(
    now.offsetFromUtc()))
```

---

The example determines the current universal and local date and time.

```
print("Local datetime: ", now.toString(Qt.ISODate))
```

The `currentDateTime()` method returns the current date and time expressed as local time. We can use the `toLocalTime()` to convert a universal time into a local time.

```
print("Universal datetime: ", now.toUTC().toString(Qt.ISODate))
```

We get the universal time with the `toUTC()` method from the date time object.

```
print("The offset from UTC is: {0} seconds".format(
    now.offsetFromUtc()))
```

The `offsetFromUtc()` gives the difference between universal time and local time in seconds.

```
$ ./utc_local.py
Local datetime: 2017-09-08T15:33:11
Universal datetime: 2017-09-08T13:33:11Z
The offset from UTC is: 7200 seconds
```

The example was run in Bratislava, which has Central European Summer Time (CEST)—UTC + 2 hours.

## 1.4 Comparing dates

Relational operators can be used to compare dates.

Listing 1.4: Comparing dates

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, Qt

d1 = QDate(2017, 4, 5)
d2 = QDate(2016, 4, 5)

if d1 < d2:
    print("{0} comes before {1}".format(d1.toString(Qt.ISODate),
                                         d2.toString(Qt.ISODate)))
else:
    print("{0} comes after {1}".format(d1.toString(Qt.ISODate),
                                       d2.toString(Qt.ISODate)))
```

---

The example compares two dates.

```
d1 = QDate(2017, 4, 5)
d2 = QDate(2016, 4, 5)
```

We define two dates.

```
if d1 < d2:
    print("{0} comes before {1}".format(d1.toString(Qt.ISODate),
                                         d2.toString(Qt.ISODate)))
else:
    print("{0} comes after {1}".format(d1.toString(Qt.ISODate),
                                       d2.toString(Qt.ISODate)))
```

We compare the two dates with a relational operator.

```
$ ./compare_dates.py
2017-04-05 comes after 2016-04-05
```

This is the output.

## 1.5 Weekday

The `dayOfWeek()` method returns a number which represents a day of a week, where 1 is Monday and 7 is Sunday.

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, QLocale

now = QDate.currentDate()

dayOfWeek = now.dayOfWeek()

print(QDate.shortDayName(dayOfWeek))
print(QDate.longDayName(dayOfWeek))

locale = QLocale(QLocale.Slovak, QLocale.Slovakia)
print(locale.toString(now, 'dddd'))
```

---

---

The example prints the weekday of the current date in local and Slovak locale.

```
dayOfWeek = now.dayOfWeek()
```

We get the day of week number with the `dayOfWeek()` method.

```
print(QDate.shortDayName(dayOfWeek))
print(QDate.longDayName(dayOfWeek))
```

We print the day name in long and short formats.

```
locale = QLocale(QLocale.Slovak, QLocale.Slovakia)
print(locale.toString(now, 'dddd'))
```

These lines print the weekday in Slovak locale.

```
$ ./weekday.py
Mon
Monday
pondelok
```

This is the output.

## 1.6 Number of days

The number of days in a particular month is returned by the `daysInMonth()` method and the number of days in a year by the `daysInYear()` method.

---

Listing 1.5: Number of days

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, Qt

now = QDate.currentDate()

d = QDate(1945, 5, 7)

print("Days in month: {}".format(d.daysInMonth()))
print("Days in year: {}".format(d.daysInYear()))
```

---

The example prints the number of days in a month and year for the chosen date.

```
$ ./days.py
Days in month: 31
Days in year: 365
```

This is the output.

## 1.7 Day of week, month, year

The `day()` method returns the day of the month, the `dayOfWeek()` the day of the week, and the `dayOfYear()` the day of the year.

---

Listing 1.6: Day of week, month, year

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, Qt

now = QDate.currentDate()

print("Day of month: {}".format(now.day()))
print("Day of week: {}".format(now.dayOfWeek()))
print("Day of year: {}".format(now.dayOfYear()))
```

---

The example prints the day of week, month, and year of the current date.

```
$ ./dayof.py
Day of month: 10
Day of week: 7
Day of year: 253
```

This is the output.

## 1.8 Difference in days

The `daysTo()` method returns the number of days from a date to another date.

---

Listing 1.7: Difference in days

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate

xmas1 = QDate(2016, 12, 24)
xmas2 = QDate(2017, 12, 24)

now = QDate.currentDate()

dayspassed = xmas1.daysTo(now)
print("{} days have passed since last XMas".format(dayspassed))

nofdays = now.daysTo(xmas2)
print("There are {} days until next XMas".format(nofdays))
```

---

The example calculates the number of days passed from the last XMas and the number of days until the next XMas.

```
$ ./xmas.py
260 days have passed since last XMas
There are 105 days until next XMas
```

This is the output.

## 1.9 Datetime arithmetic

We often need to add or subtract days, seconds, or years to a datetime value.

---

Listing 1.8: Datetime arithmetic

---

```
#!/usr/bin/python3
```

---

```

from PyQt5.QtCore import QDateTime, Qt

now = QDateTime.currentDateTime()

print("Today:", now.toString(Qt.ISODate))

print("Adding 12 days: {0}".format(
    now.addDays(12).toString(Qt.ISODate)))
print("Subtracting 22 days: {0}".format(
    now.addDays(-22).toString(Qt.ISODate)))

print("Adding 50 seconds: {0}".format(
    now.addSecs(50).toString(Qt.ISODate)))
print("Adding 3 months: {0}".format(
    now.addMonths(3).toString(Qt.ISODate)))
print("Adding 12 years: {0}".format(
    now.addYears(12).toString(Qt.ISODate)))

```

---

The example determines the current datetime and adds and subtracts days, seconds, months, and years.

```

$ ./arithmetics.py
Today: 2017-09-10T21:07:41
Adding 12 days: 2017-09-22T21:07:41
Subtracting 22 days: 2017-08-19T21:07:41
Adding 50 seconds: 2017-09-10T21:08:31
Adding 3 months: 2017-12-10T21:07:41
Adding 12 years: 2029-09-10T21:07:41

```

This is the output.

## 1.10 Daylight saving time

Daylight saving time (DST) is the practice of advancing clocks during summer months so that evening daylight lasts longer. The time is adjusted forward one hour in the beginning of spring and adjusted backward in the autumn to standard time.

---

Listing 1.9: Daylight saving time

---

```

#!/usr/bin/python3

from PyQt5.QtCore import QDateTime, QTimeZone, Qt

now = QDateTime.currentDateTime()

print("Time zone: {0}".format(now.timeZoneAbbreviation()))

if now.isDaylightTime():
    print("The current date falls into DST time")
else:
    print("The current date does not fall into DST time")

```

---

The example checks if the datetime is in the daylight saving time.

```

print("Time zone: {0}".format(now.timeZoneAbbreviation()))

```

The `timeZoneAbbreviation()` method returns the time zone abbreviation for the datetime.

```
if now.isDaylightTime():
```

The `isDaylightTime()` checks if the datetime falls in daylight saving time.

```
$ ./daylight_saving.py
Time zone: CEST
The current date falls into DST time
```

The program was executed in Bratislava, which is a city in Central Europe, in summer. Central European Summer Time (CEST) is 2 hours ahead of universal time. This time zone is a daylight saving time zone and is used in Europe and Antarctica. The standard time, which is used in winter, is Central European Time (CET).

## 1.11 Leap year

A leap year is a year containing an additional day. The reason for an extra day in the calendar is the difference between the astronomical and the calendar year. The calendar year has exactly 365 days, while the astronomical year, the time for the earth to make one revolution around the Sun, is 365.25 days. The difference is 6 hours which means that in four years time we are missing one day. Because we want to have our calendar synchronised with the seasons, we add one day to February each four years. (There are exceptions.) In the Gregorian calendar, February in a leap year has 29 days instead of the usual 28. And the year lasts 366 days instead of the usual 365.

The `isLeapYear()` method determines whether a year is a leap year.

---

Listing 1.10: Leap year

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, Qt

years = (2010, 2011, 2012, 2013, 2014, 2015, 2016)

for year in years:
    if QDate.isLeapYear(year):
        print("{0} is a leap year".format(year))
    else:
        print("{0} is not a leap year".format(year))
```

---

In the example we have a list of years. We check each year if it is a leap year.

```
years = (2010, 2011, 2012, 2013, 2014, 2015, 2016)
```

We define a tuple of years.

```
if QDate.isLeapYear(year):
```

We go through the list and determine if the given year is a leap year. The `isLeapYear()` returns a boolean True or False.

```
$ ./leapyear.py
2010 is not a leap year
```

```
2011 is not a leap year
2012 is a leap year
2013 is not a leap year
2014 is not a leap year
2015 is not a leap year
2016 is a leap year
```

This is the output.

## 1.12 Unix epoch

An epoch is an instant in time chosen as the origin of a particular era. For example in western Christian countries the time epoch starts from day 0, when Jesus was born. Another example is the French Republican Calendar which was used for twelve years. The epoch was the beginning of the Republican Era which was proclaimed on September 22, 1792, the day the First Republic was declared and the monarchy abolished.

Computers have their epochs too. One of the most popular is the Unix epoch. The Unix epoch is the time 00:00:00 UTC on 1 January 1970 (or 1970-01-01T00:00:00Z ISO 8601). The date and time in a computer is determined according to the number of seconds or clock ticks that have elapsed since the defined epoch for that computer or platform.

*Unix time* is the number of seconds elapsed since Unix epoch.

```
$ date +%s
1504875045
```

Unix date command can be used to get the Unix time. At this particular moment, 1504875045 seconds have passed since the Unix epoch.

---

Listing 1.11: Unix time

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDateTime, Qt

now = QDateTime.currentDateTime()

unix_time = now.toSecsSinceEpoch()
print(unix_time)

d = QDateTime.fromSecsSinceEpoch(unix_time)
print(d.toString(Qt.ISODate))
```

---

The example prints the Unix time and converts it back to the `QDateTime`.

```
now = QDateTime.currentDateTime()
```

First, we retrieve the current date and time.

```
unix_time = now.toSecsSinceEpoch()
```

The `toSecsSinceEpoch()` returns the Unix time.

```
d = QDateTime.fromSecsSinceEpoch(unix_time)
```

With the `fromSecsSinceEpoch()` we convert the Unix time to `QDateTime`.



```
$ ./unix_time.py
1504875266
2017-09-08T14:54:26
```

This is the output.

## 1.13 Julian day

*Julian day* refers to a continuous count of days since the beginning of the Julian Period. It is used primarily by astronomers. It should not be confused with the Julian calendar. The Julian Period started in 4713 BC. The Julian day number 0 is assigned to the day starting at noon on January 1, 4713 BC.

The Julian Day Number (JDN) is the number of days elapsed since the beginning of this period. The Julian Date (JD) of any instant is the Julian day number for the preceding noon plus the fraction of the day since that instant. (Qt does not compute this fraction.) Apart from astronomy, Julian dates are often used by military and mainframe programs.

---

Listing 1.12: Julian day

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, Qt

now = QDate.currentDate()

print("Gregorian date for today: ", now.toString(Qt.ISODate))
print("Julian day for today: ", now.toJulianDay())
```

---

In the example, we compute the Gregorian date and the Julian day for today.

```
print("Julian day for today: ", now.toJulianDay())
```

The Julian day is returned with the `toJulianDay()` method.

```
$ ./julianday.py
Gregorian date for today: 2017-09-10
Julian day for today: 2458007
```

This is the output.

With Julian day it is possible to do calculations that span centuries.

---

Listing 1.13: Battles

---

```
#!/usr/bin/python3

from PyQt5.QtCore import QDate, Qt

borodino_battle = QDate(1812, 9, 7)
slavkov_battle = QDate(1805, 12, 2)

now = QDate.currentDate()

j_today = now.toJulianDay()
j_borodino = borodino_battle.toJulianDay()
j_slavkov = slavkov_battle.toJulianDay()
```

```
d1 = j_today - j_slavkov
d2 = j_today - j_borodino

print("Days since Slavkov battle: {}".format(d1))
print("Days since Borodino battle: {}".format(d2))
```

---

The example counts the number of days passed since two historical events.

```
borodino_battle = QDate(1812, 9, 7)
slavkov_battle = QDate(1805, 12, 2)
```

We have two dates of battles of the Napoleonic era.

```
j_today = now.toJulianDay()
j_borodino = borodino_battle.toJulianDay()
j_slavkov = slavkov_battle.toJulianDay()
```

We compute the Julian days for today and for the Battles of Slavkov and Borodino.

```
d1 = j_today - j_slavkov
d2 = j_today - j_borodino
```

We compute the number of days passed since the two battles.

```
$ ./battles.py
Days since Slavkov battle: 77349
Days since Borodino battle: 74878
```

When we run this script, 77349 days have passed since the Slavkov battle, and 74878 since the Borodino battle.

## Chapter 2

# Graphics

In this chapter, we will do some low level painting on widgets. In PyQt5, `QPainter` is the class that provides highly optimized functions to do the drawing.

### 2.1 Lines

First, we draw some lines on the window. To draw lines, we use the `drawLine()` method of the painter.

Listing 2.1: Drawing Lines

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we draw lines.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter
from PyQt5.QtCore import Qt
import sys

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        self.cs = [[0 for i in range(2)] for j in range(100)]
        self.count = 0
```

```

        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Lines')
        self.show()

    def paintEvent(self, event):

        painter = QPainter()
        painter.begin(self)
        self.drawLines(painter)
        painter.end()

    def mousePressEvent(self, e):

        if e.button() == Qt.LeftButton:

            x = e.x()
            y = e.y()

            self.cs[self.count][0] = x
            self.cs[self.count][1] = y
            self.count = self.count + 1

        if e.button() == Qt.RightButton:

            self.repaint()
            self.count = 0

    def drawLines(self, painter):

        painter.setRenderHint(QPainter.Antialiasing)

        w = self.width()
        h = self.height()

        painter.eraseRect(0, 0, w, h)

        for i in range(self.count):
            for j in range(self.count):
                painter.drawLine(self.cs[i][0], self.cs[i][1],
                                self.cs[j][0], self.cs[j][1])

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

We randomly click several times with a left mouse button on the client area of the window. Then we right click on the window. The *x* and *y* coordinates of each of the mouse clicks are saved. All points are connected with a line.

```
self.cs = [[0 for i in range(2)] for j in range(100)]
```

We initiate a list of 100 two member lists. These store our mouse click coordinates.

```
self.count = 0
```

The `count` variable stores the number of clicks we have done on the window.

```
def paintEvent(self, event):  
  
    painter = QtGui.QPainter()  
    painter.begin(self)  
    self.drawLines(painter)  
    painter.end()
```

The drawing of the lines is delegated to the `drawLines()` method.

```
def mousePressEvent(self, e):  
    ...
```

In order to work with the mouse, we must reimplement the `mousePressEvent()` method.

```
if e.button() == Qt.LeftButton:  
  
    x = e.x()  
    y = e.y()  
  
    self.cs[self.count][0] = x  
    self.cs[self.count][1] = y  
    self.count = self.count + 1
```

When we click with a left mouse button, we get the `x` and `y` coordinates, store them in the coordinates list, and increase the counter.

```
if e.button() == Qt.RightButton:  
  
    self.repaint()  
    self.count = 0
```

Clicking with the right mouse button, the window is repainted and the counter is reset to zero.

```
painter.setRenderHint(QPainter.Antialiasing)
```

By setting the antialiasing rendering hint, the drawing of the lines will be smooth.

```
w = self.width()  
h = self.height()  
  
painter.eraseRect(0, 0, w, h)
```

We get the size of the window and erase the previous drawing.

```
for i in range(self.count):  
    for j in range(self.count):  
        painter.drawLine(self.cs[i][0], self.cs[i][1],  
                          self.cs[j][0], self.cs[j][1])
```

Finally, based on the coordinates that we have saved, we draw the lines with the `drawLine()` method.

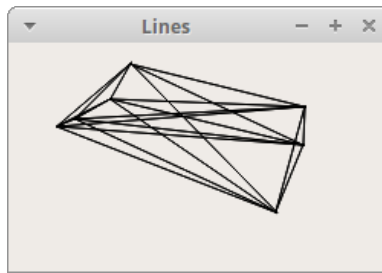


Figure 2.1: Lines

## 2.2 Drawing text

In this code example, we draw lyrics on the window. Linux users might need to install a package containing the Purisa font. On Debian systems, the package name is `fonts-tlwg-purisa`.

Windows users need to copy the Purisa font to the Fonts directory of the Windows system directory. The font is included in the ZIP file that you have purchased. Another option is to use some other font that you like.

Listing 2.2: Drawing Text

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we draw lyrics on the
window in a Purisa font.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QFont
import sys

lyrics = [
    "Most relationships seem so transitory",
    "They're all good but not the permanent one",
    "Who doesn't long for someone to hold",
    "Who knows how to love without being told",
    "Somebody tell me why I'm on my own",
    "If there's a soulmate for everyone"
]

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()
```

```

def initUI(self):

    self.setGeometry(300, 300, 430, 240)
    self.setWindowTitle('Soulmate')
    self.show()

def paintEvent(self, event):

    painter = QPainter()
    painter.begin(self)
    self.drawLyrics(painter)
    painter.end()

def drawLyrics(self, painter):

    painter.setFont(QFont('Purisa', 11))

    painter.drawText(20, 30, lyrics[0])
    painter.drawText(20, 60, lyrics[1])
    painter.drawText(20, 120, lyrics[2])
    painter.drawText(20, 150, lyrics[3])
    painter.drawText(20, 180, lyrics[4])
    painter.drawText(20, 210, lyrics[5])

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

We draw a part of the Natasha Bedingfield's song on the window. We use a specific Purisa font.

```

lyrics = [
    "Most relationships seem so transitory",
    "They're all good but not the permanent one",
    "Who doesn't long for someone to hold",
    "Who knows how to love without being told",
    "Somebody tell me why I'm on my own",
    "If there's a soulmate for everyone"
]

```

This is the text to display.

```

def paintEvent(self, event):

    painter = QPainter()
    painter.begin(self)
    self.drawLyrics(painter)
    painter.end()

```

All the painting is done inside the `paintEvent()` method. The painter object is initiated. The actual drawing is delegated to the `drawLyrics()` user method.

```

painter.setFont(QFont('Purisa', 11))

```

By calling the `setFont()` method, we draw in a chosen font. In our case, it is 11 points Purisa font.

```
painter.drawText(20, 30, lyrics[0])
```

The `drawText()` method draws text on the window. It uses the font that we have specified earlier.

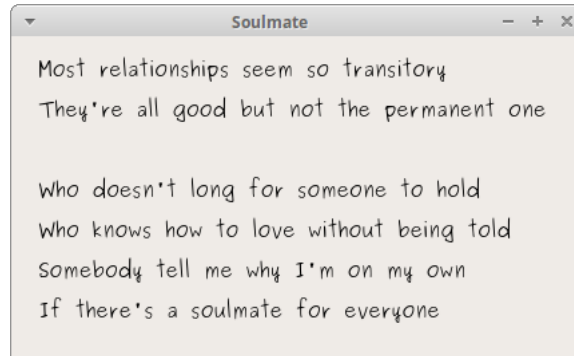


Figure 2.2: Lyrics on the window

## 2.3 Grayscale image

Grayscale image, also known as black-and-white, are composed exclusively of shades of gray, varying from black at the weakest intensity to white at the strongest. In the following example, we programatically create a grayscale image.

Listing 2.3: Grayscale Image

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a grayscale image.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QImage, qGray, qAlpha, qRgba
import sys

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()
```



```

def initUI(self):

    self.sid = QImage("smallsid.jpg")

    self.w = self.sid.width()
    self.h = self.sid.height()

    self.setGeometry(200, 200, 320, 150)
    self.setWindowTitle('Sid')
    self.show()

def paintEvent(self, event):

    painter = QPainter()
    painter.begin(self)
    self.drawImages(painter)
    painter.end()

def grayScale(self, image):

    for i in range(self.w):
        for j in range(self.h):

            c = image.pixel(i, j)
            gray = qGray(c)
            alpha = qAlpha(c)
            image.setPixel(i, j,
                           qRgb(gray, gray, gray, alpha))

    return image

def drawImages(self, painter):

    painter.drawImage(5, 15, self.sid)
    painter.drawImage(self.w+10, 15,
                      self.grayScale(self.sid.copy()))

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

We have a color JPG image. The image is drawn on the window. We create a copy of the image, convert it to grayscale and draw it on the window next to the original image.

```

def initUI(self):

    self.sid = QImage("smallsid.jpg")

    self.w = self.sid.width()
    self.h = self.sid.height()

    self.setGeometry(200, 200, 320, 150)
    self.setWindowTitle('Sid')
    self.show()

```

The `initUI()` method loads the image and gets the width and the height of the image.

```
def grayScale(self, image):  
  
    for i in range(self.w):  
        for j in range(self.h):  
  
            c = image.pixel(i, j)  
            gray = qGray(c)  
            alpha = qAlpha(c)  
            image.setPixel(i, j,  
                           qRgb(gray, gray, gray, alpha))  
  
    return image
```

The `grayScale()` method transforms the image to grayscale and returns it. We go through all pixels of the image in two for loops. The `pixel()` method returns the pixel in question. We use the `qGray()` global method to get the gray value of a specific pixel. Similarly, we get the alpha value. Finally, we modify the pixel with the `setPixel()` method. We use the gray value for the red, green, and blue parts of the color.

```
def drawImages(self, painter):  
  
    painter.drawImage(5, 15, self.sid)  
    painter.drawImage(self.w+10, 15,  
                      self.grayScale(self.sid.copy()))
```

Inside the `drawImages()` method, we draw the images. Notice that we send a copy of the original image to the `grayScale()` method. Otherwise, we would change the original image.



Figure 2.3: Grayscale image

## 2.4 Donut

In the following example we create a complex shape by rotating an ellipse.

---

Listing 2.4: Drawing Donut Shape

---

```
def drawDonut(self, painter):  
  
    brush = QBrush(QColor("#535353"))
```

```

painter.setPen(QPen(brush, 0.5))

painter.setRenderHint(QPainter.Antialiasing)

h = self.height()
w = self.width()

painter.translate(QPoint(w/2, h/2))

rot = 0

while rot < 360.0:

    painter.drawEllipse(-125, -40, 250, 80)
    painter.rotate(5.0)
    rot += 5.0

```

---

The shape resembles a cookie, hence the name donut.

```

brush = QBrush(QColor("#535353"))
painter.setPen(QPen(brush, 0.5))

```

The pen used to draw outlines of the shapes is of gray color. Its width is 0.5 px.

```

painter.setRenderHint(QPainter.Antialiasing)

```

We use using anti-aliasing in the example. The shapes are going to be smoother.

```

painter.translate(QPoint(w/2, h/2))

```

We translate the origin of the transformation system to the middle of the window. This makes the drawing easier.

```

while rot < 360.0:

    painter.drawEllipse(-125, -40, 250, 80)
    painter.rotate(5.0)
    rot += 5.0

```

We draw an ellipse object 72 times. Each time we rotate the ellipse by 5 degrees. The donut shape is created.

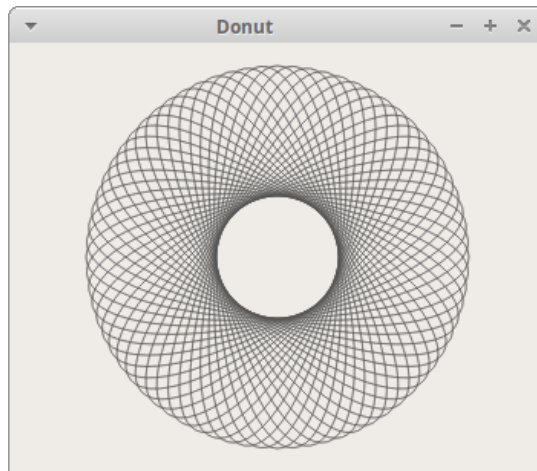


Figure 2.4: Donut Shape

## 2.5 Shapes

The PyQt5 painting API can draw various shapes, including rectangles, circles, and polygons.

Listing 2.5: Drawing Shapes

---

```
def drawShapes(self, painter):

    painter.setRenderHint(QPainter.Antialiasing)
    painter.setPen(Qt.NoPen)
    painter.setBrush(QBrush(QColor("#888888")))

    path1 = QPainterPath()

    path1.moveTo(5, 5)
    path1.cubicTo(40, 5, 50, 50, 99, 99)
    path1.cubicTo(5, 99, 50, 50, 5, 5)
    painter.drawPath(path1)

    painter.drawPie(130, 20, 90, 60, 30*16, 120*16)
    painter.drawChord(240, 30, 90, 60, 0, 16*180)
    painter.drawRoundedRect(20, 120, 80, 50, 10, 10)

    polygon = QPolygon()
    polygon.append(QPoint(130, 140))
    polygon.append(QPoint(180, 170))
    polygon.append(QPoint(180, 140))
    polygon.append(QPoint(220, 110))
    polygon.append(QPoint(140, 100))

    painter.drawPolygon(polygon)

    painter.drawRect(250, 110, 60, 60)

    baseline = QPointF(20, 250)
    font = QFont("Georgia", 55)
```

```

path2 = QPainterPath()
path2.addText(baseline, font, "Q")
painter.drawPath(path2)

painter.drawEllipse(140, 200, 60, 60)
painter.drawEllipse(240, 200, 90, 60)

```

In the above code example, we draw various shapes on the window.

```

painter.setPen(Qt.NoPen)
painter.setBrush(QBrush(QColor("#888888")))

```

We do not use a pen. We paint only the insides of the shapes with a gray color.

```

path1 = QPainterPath()

path1.moveTo(5, 5)
path1.cubicTo(40, 5, 50, 50, 99, 99)
path1.cubicTo(5, 99, 50, 50, 5, 5)
painter.drawPath(path1)

```

The `QPainterPath` is a class used to create complex shapes. We create a shape using bezier curves.

```

painter.drawPie(130, 20, 90, 60, 30*16, 120*16)
painter.drawChord(240, 30, 90, 60, 0, 16*180)
painter.drawRoundedRect(20, 120, 80, 50, 10, 10)

```

These lines draw a pie, a chord, and a rounded rectangle.

```

polygon = QPolygon()
polygon.append(QPoint(130, 140))
polygon.append(QPoint(180, 170))
polygon.append(QPoint(180, 140))
polygon.append(QPoint(220, 110))
polygon.append(QPoint(140, 100))

painter.drawPolygon(polygon)

```

We draw a polygon consisting of five points.

```

painter.drawRect(250, 110, 60, 60)

```

This line creates a simple rectangle.

```

baseline = QPointF(20, 250)
font = QFont("Georgia", 55)
path2 = QPainterPath()
path2.addText(baseline, font, "Q")
painter.drawPath(path2)

```

Here we create a path based on a font character.

```

painter.drawEllipse(140, 200, 60, 60)
painter.drawEllipse(240, 200, 90, 60)

```

The `drawEllipse()` method can be used to draw an ellipse and a circle as well. The circle is a special case of an ellipse.

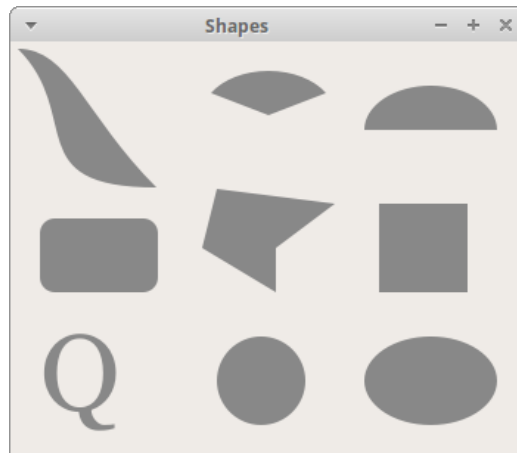


Figure 2.5: Shapes

## 2.6 Transparent rectangles

Transparency is the quality of being able to see through a material. The easiest way to understand transparency is to imagine a piece of glass or water. Technically, the rays of light can go through the glass and this way we can see objects behind the glass.

In computer graphics, we can achieve transparency effects using alpha compositing. Alpha compositing is the process of combining an image with a background to create the appearance of partial transparency. The composition process uses an alpha channel.

Listing 2.6: Transparent Rectangles

```
def drawRectangles(self, painter):  
    for i in range(1, 11):  
        painter.setOpacity(i*0.1)  
        painter.fillRect(50*i, 20, 40, 40,  
                        Qt.darkGray)
```

These lines draw ten rectangles with different levels of transparency. The transparency is controlled with the `setOpacity()` method. The lower is the opacity, the higher is the transparency. The `fillRect()` method draws the insides of the rectangles with a dark gray color, taking the level of transparency into account.

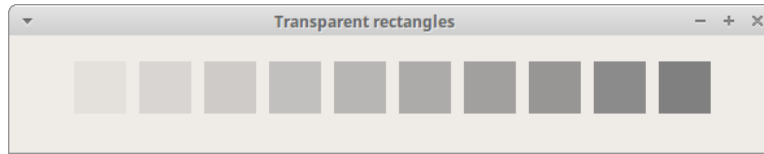


Figure 2.6: Transparent rectangles

## 2.7 Gradients

In computer graphics, a gradient is a smooth blending of shades from light to dark or from one color to another. In drawing and paint programs, gradients are used to create colorful backgrounds and special effects as well as to simulate lights and shadows. There are two types of gradients: linear gradients and radial gradients.

A linear gradient is defined by the `QLinearGradient` class.

Listing 2.7: Linear Gradients

---

```
def drawGradients(self, painter):

    grad1 = QLinearGradient(0, 20, 0, 110)

    grad1.setColorAt(0.1, Qt.black)
    grad1.setColorAt(0.5, Qt.yellow)
    grad1.setColorAt(0.9, Qt.black)

    painter.fillRect(20, 20, 300, 90, grad1)

    grad2 = QLinearGradient(0, 55, 250, 0)

    grad2.setColorAt(0.2, Qt.black)
    grad2.setColorAt(0.5, Qt.red)
    grad2.setColorAt(0.8, Qt.black)

    painter.fillRect(20, 140, 300, 100, grad2)
```

---

The example draws two rectangles filled with linear gradients.

```
grad1 = QLinearGradient(0, 20, 0, 110)
```

Here we create a linear gradient. The parameters specify the line along which we draw the gradient. Here it is a horizontal line.

```
grad1.setColorAt(0.1, Qt.black)
grad1.setColorAt(0.5, Qt.yellow)
grad1.setColorAt(0.9, Qt.black)
```

We define color stops to produce our gradient pattern. In this case, the gradient is a blending of black and yellow colors. By adding two black and one yellow stop, we create a horizontal gradient pattern. We begin with black color which stops at 1/10 of the size. Then we begin gradually painting in yellow which culminates at the center of the shape. The yellow color stops at 9/10 of the size where we begin painting in black again, until the end.

```
painter.fillRect(20, 20, 300, 90, grad1)
```

A rectangle is filled with the gradient.

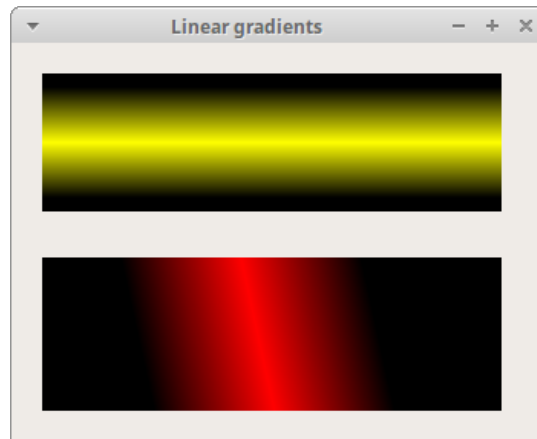


Figure 2.7: Linear gradients

A radial gradient is a smooth blending of colors or shades of colors between a circle and a focal point. A radial gradient is defined by the `QRadialGradient` class.

Listing 2.8: Radial Gradients

```
def drawGradients(self, painter):  
  
    painter.setRenderHint(QPainter.Antialiasing)  
    painter.setPen(Qt.NoPen)  
  
    gr1 = QRadialGradient(20.0, 20.0, 110.0)  
    painter.setBrush(gr1)  
    painter.drawEllipse(20.0, 20.0, 100.0, 100.0)  
  
    gr2 = QRadialGradient(190.0, 70.0, 50.0, 190.0, 70.0)  
    gr2.setColorAt(0.2, Qt.yellow)  
    gr2.setColorAt(0.7, Qt.black)  
    painter.setBrush(gr2)  
    painter.drawEllipse(140.0, 20.0, 100.0, 100.0)
```

Two radial gradients are drawn in the example.

```
gr1 = QRadialGradient(20.0, 20.0, 110.0)  
painter.setBrush(gr1)  
painter.drawEllipse(20.0, 20.0, 100.0, 100.0)
```

The parameters to the `QRadialGradient` class are the x and y coordinates of the center point and the radius. The focal point lies at the center of the circle.

```
gr2 = QRadialGradient(190.0, 70.0, 50.0, 190.0, 70.0)  
gr2.setColorAt(0.2, Qt.yellow)  
gr2.setColorAt(0.7, Qt.black)  
painter.setBrush(gr2)
```



```
painter.drawEllipse(140.0, 20.0, 100.0, 100.0)
```

In the second gradient, the focal point lies outside the center of the circle. The blending uses yellow and black colors.

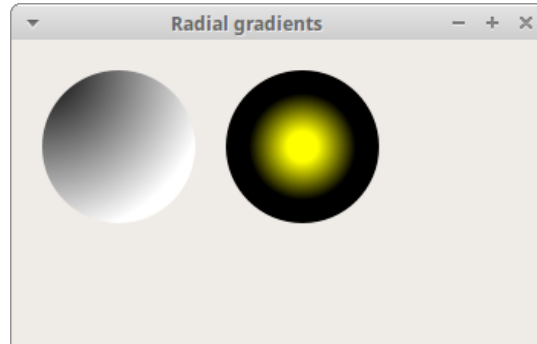


Figure 2.8: Radial gradients

## 2.8 Waiting effect

In this example, we use transparency to create a waiting demo. We draw eight lines that gradually fade out creating an illusion that a line is moving. Such effects are often used to inform users that a lengthy task is going on behind the scenes; for instance, when streaming video over the Internet.

Listing 2.9: Waiting Effect

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a waiting
effect.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QPen
from PyQt5.QtCore import Qt
import sys

trs = (
    ( 0.0, 0.15, 0.30, 0.5, 0.65, 0.80, 0.9, 1.0 ),
    ( 1.0, 0.0, 0.15, 0.30, 0.5, 0.65, 0.8, 0.9 ),
    ( 0.9, 1.0, 0.0, 0.15, 0.3, 0.5, 0.65, 0.8 ),
    ( 0.8, 0.9, 1.0, 0.0, 0.15, 0.3, 0.5, 0.65 ),
    ( 0.65, 0.8, 0.9, 1.0, 0.0, 0.15, 0.3, 0.5 ),
    ( 0.5, 0.65, 0.8, 0.9, 1.0, 0.0, 0.15, 0.3 ),
```

```

        ( 0.3, 0.5, 0.65, 0.8, 0.9, 1.0, 0.0, 0.15 ),
        ( 0.15, 0.3, 0.5, 0.65, 0.8, 0.9, 1.0, 0.0, )
    )

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        self.count = 0
        self.timerId = self.startTimer(105)

        self.setGeometry(300, 300, 250, 200)
        self.setWindowTitle('Waiting')
        self.show()

    def paintEvent(self, event):

        painter = QPainter()

        painter.begin(self)
        painter.setRenderHint(QPainter.Antialiasing)
        self.drawLines(painter)
        painter.end()

    def drawLines(self, painter):

        pen = QPen()
        pen.setWidth(3)
        pen.setCapStyle(Qt.RoundCap)

        w = self.width()
        h = self.height()

        painter.translate(w/2, h/2)
        painter.setPen(pen)

        for i in range(8):

            painter.setOpacity(trs[self.count%8][i])
            painter.drawLine(0.0, -10.0, 0.0, -40.0)
            painter.rotate(45)

    def timerEvent(self, event):

        self.count = self.count + 1
        self.repaint()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

In the example we draw eight lines with eight different alpha values.

```
trs = (  
    ( 0.0, 0.15, 0.30, 0.5, 0.65, 0.80, 0.9, 1.0 ),  
    ( 1.0, 0.0, 0.15, 0.30, 0.5, 0.65, 0.8, 0.9 ),  
    ...  
)
```

This is a collection of transparency values. There are eight rows, each for one position. Each of the eight lines will continuously use these values.

```
def initUI(self):  
  
    self.count = 0  
    self.timerId = self.startTimer(105)  
  
    self.setGeometry(300, 300, 250, 200)  
    self.setWindowTitle('Waiting')  
    self.show()
```

The first two lines initiate the `count` variable and start a timer.

```
pen = QPen()  
pen.setWidth(3)  
pen.setCapStyle(Qt.RoundCap)
```

We make the lines a bit thicker so that they are more visible. We draw the lines with rounded caps. Lines with rounded caps look better.

```
for i in range(8):  
  
    painter.setOpacity(trs[self.count%8][i])  
    painter.drawLine(0.0, -10.0, 0.0, -40.0)  
    painter.rotate(45)
```

In this loop, we set the opacity value. We draw the line and rotate it. This creates an illusion of a moving and fading line.

```
def timerEvent(self, event):  
  
    self.count = self.count + 1  
    self.repaint()
```

Each time the timer event is called, we increase the `count` value and repaint the window area.

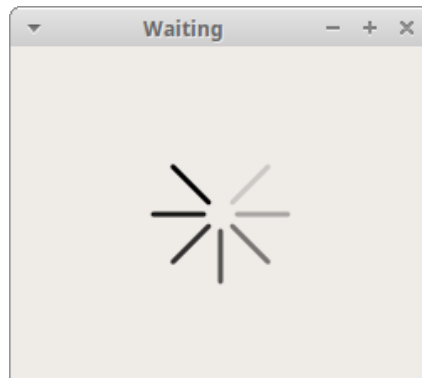


Figure 2.9: Waiting

## 2.9 Puff effect

A puff effect is a very common effect which we can often see in animations on the web. The next example displays a growing centered text that gradually fades out from some point.

Listing 2.10: Puff Effect

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a growing
and fading text effect.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import (QPainter, QBrush, QColor,
                        QPen, QFont, QFontMetrics)
from PyQt5.QtCore import QPoint
import sys

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        self.h = 1
        self.opacity = 1.0
```

```

        self.timerId = self.startTimer(15)

        self.setGeometry(300, 300, 350, 280)
        self.setWindowTitle('Puff')
        self.show()

    def paintEvent(self, event):

        painter = QPainter()
        painter.begin(self)
        self.doStep(painter)
        painter.end()

    def doStep(self, painter):

        text = "ZetCode"

        brush = QBrush(QColor("#575555"))
        painter.setPen(QPen(brush, 1))

        f = QFont("Verdana", self.h)
        f.setWeight(QFont.DemiBold)

        fm = QFontMetrics(f)
        textWidth = fm.width(text)

        painter.setFont(f)

        if self.h > 10:
            self.opacity -= 0.01
            painter.setOpacity(self.opacity)

        if self.opacity <= 0:
            self.killTimer(self.timerId)

        h = self.height()
        w = self.width()

        painter.translate(QPoint(w/2, h/2))
        painter.drawText(-textWidth/2, 0, text)

    def timerEvent(self, event):

        self.h = self.h + 1
        self.repaint()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

This is the code for the Puff effect.

```

def initUI(self):

    self.h = 1
    self.opacity = 1.0
    self.timerId = self.startTimer(15)

```

```

self.setGeometry(300, 300, 350, 280)
self.setWindowTitle('Puff')
self.show()

```

In the `initUI()` method, we set two variables and start the timer. The first variable is the text height, the second is the opacity. Each 15 ms a timer event is generated and the `timerEvent()` method is called.

```

fm = QFontMetrics(f)
textWidth = fm.width(text)

```

`QFontMetrics` methods calculate the size of characters and strings for a given font. We want to show a centered text on the window. Therefore, we need to get the width of the text.

```

if self.h > 10:
    self.opacity -= 0.01
    painter.setOpacity(self.opacity)

```

If the font height is greater than 10 points, we gradually decrease the opacity. The text fades away. The value of the opacity is in the range 0.0 to 1.0, where 0.0 is fully transparent and 1.0 is fully opaque. Opacity set on the painter will apply to all drawing operations individually.

```

if self.opacity <= 0:
    self.killTimer(self.timerId)

```

If the text completely fades away, we kill the timer.

```

h = self.height()
w = self.width()

```

We get the width and height of the window.

```

painter.translate(QPoint(w/2, h/2))
painter.drawText(-textWidth/2, 0, text)

```

We draw the text in the center of the window. To accomplish this, we need three values: window height and width and the text width.

```

def timerEvent(self, event):

    self.h = self.h + 1
    self.repaint()

```

Inside the `timerEvent()` method, we increase the font height and repaint the window.

## 2.10 Hit test

Sometimes we need to know if we have clicked inside a shape with a mouse pointer. The `QRect` class has a `contains()` method which will do the job.

Listing 2.11: Hit Test

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''

```

*ZetCode Advanced PyQt5 tutorial*

*In this example, we we work with a thread.  
We click on the area of the rectangle and  
the rectangle starts to fade away.*

*Author: Jan Bodnar*

*Website: zetcode.com*

*Last edited: August 2017*

*'''*

```
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QBrush
from PyQt5.QtCore import Qt, QThread, pyqtSignal, QThread, QRect
import time
import sys
```

```
class MyThread(QThread):
```

```
    fading = pyqtSignal(float, name='fading')
```

```
    def __init__(self):
        super().__init__()
```

```
        self.alpha = 1
```

```
    def run(self):
```

```
        print("Thread started")
```

```
        while self.alpha >= 0:
```

```
            self.alpha -= 0.01
            self.fading.emit(self.alpha)
            time.sleep(0.1)
```

```
        print("Thread ended")
```

```
class Example(QWidget):
```

```
    def __init__(self):
        super().__init__()
```

```
        self.initUI()
```

```
    def initUI(self):
```

```
        self.rect = QRect(20, 20, 80, 80)
        self.alpha = 1
        self.m = MyThread()
```

```
        self.m.fading.connect(self.fade)
```

```
        self.setGeometry(300, 300, 250, 150)
        self.setWindowTitle('Hit test')
        self.show()
```

```
    def paintEvent(self, event):
```

```

        painter = QPainter()
        painter.begin(self)
        self.drawRectangle(painter)
        painter.end()

    def fade(self, alpha):

        self.alpha = alpha
        self.repaint()

    def mousePressEvent(self, e):

        if e.button() == Qt.LeftButton \
            and self.alpha==1:

            x = e.x()
            y = e.y()

            if self.rect.contains(x, y):
                if not self.m.isRunning():
                    self.m.start()

    def drawRectangle(self, painter):

        painter.setOpacity(self.alpha)
        painter.setBrush(QBrush(Qt.black))
        painter.setPen(Qt.NoPen)
        painter.drawRect(self.rect)

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

We draw a black rectangle on the window. If we click inside the rectangle, it starts to fade away. We utilize the `QThread` class for fading out the rectangle.

```

class MyThread(QThread):

    fading = pyqtSignal(float, name='fading')

    def __init__(self):
        super(MyThread, self).__init__()

        self.alpha = 1

```

A thread is a specific executional unit inside a program. Our thread decreases the `alpha` value which controls the transparency of the rectangle. A custom fading signal is defined in the class. Note that we do not modify a variable in the main program; instead, we send a computed value to the main program in a custom signal.

```

def run(self):

    print("Thread started")

    while self.alpha >= 0:

```



```

        self.alpha -= 0.01
        self.fading.emit(self.alpha)
        time.sleep(0.1)

    print("Thread ended")

```

When the thread is started, the `run()` method is being called. Note that the method is called only once. To do some repetitive work, we must create a loop. Inside the loop, we do three things. We decrease the `alpha` value. We emit a signal with the `alpha` value and stop the thread for one millisecond.

```

self.m = MyThread()
self.m.fading.connect(self.fade)

```

An instance of the thread is created. When we receive a fading signal from the thread, we call the `fade()` method.

```

def fade(self, alpha):

    self.alpha = alpha
    self.repaint()

```

The `fade()` method stores the current `alpha` value and repaints the window.

```

x = e.x()
y = e.y()

if self.rect.contains(x, y):
    if not self.m.isRunning():
        self.m.start()

```

In the body of the `mousePressEvent()`, we get the `x` and `y` coordinates of a mouse click. If the coordinates are inside the area of our rectangle, we start a thread. We use the `contains()` method to figure out if we have clicked inside the rectangle.

```

def drawRectangle(self, painter):

    painter.setOpacity(self.alpha)
    painter.setBrush(QBrush(Qt.black))
    painter.setPen(Qt.NoPen)
    painter.drawRect(self.rect)

```

The rectangle is drawn. We take the transparency value into account.

## 2.11 Image Reflection

In the next example we show a reflected image. This effect makes an illusion an image being reflected in water.

---

Listing 2.12: Image Reflection

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

```

*In this example, we create a reflected image.*

*Author: Jan Bodnar*

*Website: [zetcode.com](http://zetcode.com)*

*Last edited: August 2017*

*'''*

```
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QColor, QImage, QLinearGradient
from PyQt5.QtCore import Qt, QRect
import sys
```

```
class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.loadImage()
        self.initUI()
        self.createReflectedImage()

    def loadImage(self):

        self.img = QImage("slanec.png")

        if self.img.isNull():
            print("Error loading image")
            sys.exit(1)

    def initUI(self):

        self.iw = self.img.width()
        self.ih = self.img.height()

        self.setGeometry(200, 200, 300, 400)
        self.setWindowTitle('Reflection')
        self.show()

    def createReflectedImage(self):

        self.refImage = QImage(self.iw, self.ih,
                                QImage.Format_ARGB32)

        painter = QPainter()
        painter.begin(self.refImage)

        painter.drawImage(0, 0, self.img)

        mode = QPainter.CompositionMode_DestinationIn
        painter.setCompositionMode(mode)

        gradient = QLinearGradient(self.iw/2, 0,
                                    self.iw/2, self.ih)

        gradient.setColorAt(1, QColor(0, 0, 0))
        gradient.setColorAt(0, Qt.transparent)
```

```

        painter.fillRect(0, 0, self.iw, self.ih, gradient)

    painter.end()

def paintEvent(self, event):

    painter = QPainter()
    painter.begin(self)
    self.drawImages(painter)
    painter.end()

def drawImages(self, painter):

    w = self.width()
    h = self.height()

    gradient = QLinearGradient(w/2, 0, w/2, h)

    gradient.setColorAt(0, Qt.black)
    gradient.setColorAt(1, Qt.darkGray)

    painter.fillRect(0, 0, w, h, gradient)

    gap = 30

    rect = QRect(25, 15, self.iw, self.ih)
    painter.drawImage(rect, self.img)
    painter.translate(0, 2*self.ih + gap)
    painter.scale(1, -1)

    painter.drawImage(25, 0, self.refImage)

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

This is a reflected image code example. The original image is drawn on the window initially. Below that image, we draw a reflected image which is flipped and whose transparency grows gradually.

```

def loadImage(self):

    self.img = QImage("slanec.png")

    if self.img.isNull():
        print("Error loading image")
        sys.exit(1)

```

In the `loadImage()` method, we load an image and do some error checking.

```

self.iw = self.img.width()
self.ih = self.img.height()

```

The width and height of the image is determined.

```

def createReflectedImage(self):

    self.refImage = QImage(self.iw, self.ih,
        QImage.Format_ARGB32)

```

```

        painter = QPainter()
        painter.begin(self.refImage)
    ...

```

In the `createReflectedImage()` method, we create an image that will serve as a reflection of the original image. A new `QImage` is formed. We will be painting into this image.

```

painter.drawImage(0, 0, self.img)

mode = QPainter.CompositionMode_DestinationIn
painter.setCompositionMode(mode)

gradient = QLinearGradient(self.iw/2, 0,
                           self.iw/2, self.ih)

gradient.setColorAt(1, QColor(0, 0, 0))
gradient.setColorAt(0, Qt.transparent)

painter.fillRect(0, 0, self.iw, self.ih, gradient)

```

We draw a picture and a linear gradient. The two objects are blended using a `QPainter.CompositionMode_DestinationIn` composition mode.

```

def drawImages(self, painter):
    ...

```

Inside the `drawImages()` method, we draw both images and paint the background with a linear gradient.

```

w = self.width()
h = self.height()

gradient = QLinearGradient(w/2, 0, w/2, h)

gradient.setColorAt(0, Qt.black)
gradient.setColorAt(1, Qt.darkGray)

painter.fillRect(0, 0, w, h, gradient)

```

The background of the window is filled with a specific linear gradient. The gradient is a blending of a black and dark gray colors.

```

gap = 30

```

A gap is a space between the two images.

```

rect = QRect(25, 15, self.iw, self.ih)
painter.drawImage(rect, self.img)

```

A ruin of a castle is drawn on the window.

```

painter.translate(0, 2*self.ih + gap)
painter.scale(1, -1)

```

The scaling operation flips the image. It also moves the image up. Therefore, we need to perform a translation operation.

```

painter.drawImage(25, 0, self.refImage)

```

The reflected image is drawn below the original image.



Figure 2.10: Image reflection

## 2.12 Clipping

Clipping is restricting of drawing to a certain area. This is done for efficiency reasons and to create various effects. In our example, we will be clipping two shapes: a rectangle and a circle.

Listing 2.13: Clipping

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example demonstrates clipping.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QBrush, QColor
from PyQt5.QtCore import Qt, QRect
import sys

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()
```

```

def initUI(self):

    self.rotate = 1
    self.pos_x = 8
    self.pos_y = 8
    self.radius = 60

    self.delta = [1, 1]
    self.timerId = self.startTimer(15)

    self.setGeometry(300, 300, 350, 250)
    self.setWindowTitle('Clipping')
    self.show()

def paintEvent(self, event):

    painter = QPainter()
    painter.begin(self)
    self.drawObjects(painter)
    painter.end()

def drawObjects(self, painter):

    painter.setRenderHint(QPainter.Antialiasing)

    w = self.width()
    h = self.height()
    rect = QRect(-100, -40, 200, 80)

    painter.translate(w/2, h/2)
    painter.rotate(self.rotate)
    painter.drawRect(rect)

    brush = QBrush(QColor(110, 110, 110))
    painter.setBrush(brush)

    painter.setClipRect(rect)

    painter.resetTransform()
    painter.drawEllipse(self.pos_x, self.pos_y,
                        self.radius, self.radius)

    painter.setBrush(Qt.NoBrush)

    painter.setClipping(False)
    painter.drawEllipse(self.pos_x, self.pos_y,
                        self.radius, self.radius)

def timerEvent(self, event):

    self.step()
    self.repaint()

def step(self):

    w = self.width()

```

```

        h = self.height()

        if self.pos_x < 0:
            self.delta[0] = 1

        elif self.pos_x > w - self.radius:
            self.delta[0] = -1

        if self.pos_y < 0:
            self.delta[1] = 1

        elif self.pos_y > h - self.radius:
            self.delta[1] = -1

        self.pos_x += self.delta[0]
        self.pos_y += self.delta[1]

        self.rotate = self.rotate + 1

def main():

    app = QApplication([])
    ex = Example()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

---

We have two moving objects: a rectangle and a circle. The rectangle rotates around its center point. The circle moves and bounces off the borders. When these shapes overlap, the resulting area is filled with color.

```
self.delta = [1, 1]
```

In the `initUI()` method, we have a `delta` variable. It is a list of two values. They control the movement of the circle in the x and y direction.

```

painter.translate(w/2, h/2)
painter.rotate(self.rotate)
painter.drawRect(rect)

```

We move the coordinate system into the middle of the window. We do the rotation and draw the rectangle.

```

brush = QtGui.QBrush(QtGui.QColor(110, 110, 110))
painter.setBrush(brush)

```

We set a brush which is applied to the overlapping area.

```
painter.setClipRect(rect)
```

The `setClipRect()` enables clipping and sets the clip region to the given rectangle.

```
painter.resetTransform()
```

While drawing the rotating rectangle, we have done some transformations. Now we reset those. The next drawing is independent from the previous one.

```
painter.drawEllipse(self.pos_x, self.pos_y,
```

```
self.radius, self.radius)
```

We fill the restricted area with color by calling the `drawEllipse()` method. Only the part of the ellipse that overlaps with the rectangle is filled.

```
painter.setBrush(Qt.NoBrush)

painter.setClipping(False)
painter.drawEllipse(self.pos_x, self.pos_y,
                    self.radius, self.radius)
```

We remove the brush and reset the clipping. Finally, we draw the ellipse object.

```
if self.pos_x < 0:
    self.delta[0] = 1

elif self.pos_x > w - self.radius:
    self.delta[0] = -1
```

If the circle hits the left or the right border, we change the direction of its movement.

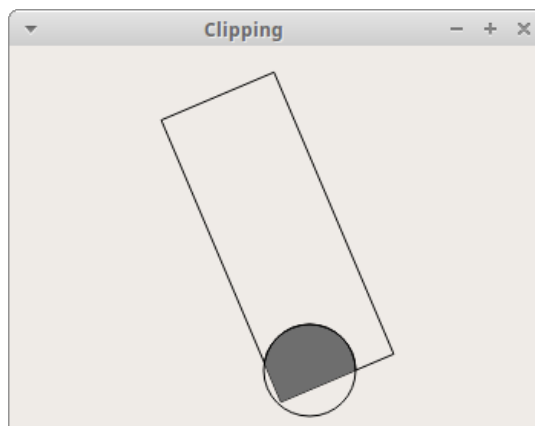


Figure 2.11: Clipping

## 2.13 Moving star

In the following example, we create a rotating and scaling star. The star rotates, and grows or shrinks.

Listing 2.14: Moving Star

```
#!/usr/bin/python3

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a rotating
and scaling star.
```



*Author: Jan Bodnar*  
*Website: zetcode.com*  
*Last edited: August 2017*  
,,

```
from PyQt5.QtWidgets import QWidget, QApplication
from PyQt5.QtGui import QPainter, QPainterPath, QBrush
from PyQt5.QtCore import Qt
import sys

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()
        self.initVariables()

    def initVariables(self):

        self.points = [
            [0, 85], [75, 75], [100, 10], [125, 75],
            [200, 85], [150, 125], [160, 190], [100, 150],
            [40, 190], [50, 125], [0, 85]
        ]

        self.angle = 0
        self.scale = 1
        self.delta = 0.01
        self.timerId = self.startTimer(15)

    def initUI(self):

        self.setGeometry(300, 300, 350, 250)
        self.setWindowTitle('Moving Star')
        self.show()

    def paintEvent(self, event):

        painter = QPainter()
        painter.begin(self)
        self.drawStar(painter)
        painter.end()

    def drawStar(self, painter):

        painter.setRenderHint(QPainter.Antialiasing)

        h = self.height()
        w = self.width()

        painter.translate(w/2, h/2)
        painter.rotate(self.angle)
        painter.scale(self.scale, self.scale)

        path = QPainterPath()
        path.moveTo(self.points[0][0], self.points[0][1])
```

```

        for i in range(len(self.points)):
            path.lineTo(self.points[i][0], self.points[i][1])

        brush = QBrush(Qt.SolidPattern)
        painter.fillPath(path, brush)

    def timerEvent(self, event):

        if self.scale < 0.01:
            self.delta = -self.delta
        elif self.scale > 0.99:
            self.delta = -self.delta

        self.scale += self.delta
        self.angle += 1

        self.repaint()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

We have a rotating and scaling star.

```

self.points = [
    [0, 85], [75, 75], [100, 10], [125, 75],
    [200, 85], [150, 125], [160, 190], [100, 150],
    [40, 190], [50, 125], [0, 85]
]

```

These points are used to create a star object.

```

self.angle = 0
self.scale = 1
self.delta = 0.01

```

The `self.angle` variable is controlling the rotation of the star. The `self.scale` is responsible for the current size of the star. The `self.delta` is one step in the shrinking/growing of the star.

```

painter.translate(w/2, h/2)
painter.rotate(self.angle)
painter.scale(self.scale, self.scale)

```

The star shape is translated to the center of the window, rotated, and scaled.

```

path = QPainterPath()
path.moveTo(self.points[0][0], self.points[0][1])

for i in range(len(self.points)):
    path.lineTo(self.points[i][0], self.points[i][1])

brush = QBrush(Qt.SolidPattern)
painter.fillPath(path, brush)

```

The star object is created using the `QPainterPath` class.

```

if self.scale < 0.01:

```

```
        self.delta = -self.delta  
elif self.scale > 0.99:  
    self.delta = -self.delta  
  
self.scale += self.delta
```

These lines control the growing/shrinking of the star.

## Chapter 3

# The Graphics View framework

In the previous chapter, we have covered low level painting with the help of the `QPainter` class. The Graphics View framework is a high level graphics framework. The framework is a set of tools to help the developers create graphics more quickly and efficiently. It has some useful built-in capabilities like zooming or animation. It efficiently renders thousands of 2D objects onscreen.

The framework consists of three basic classes:

- `QGraphicsView`
- `QGraphicsScene`
- `QGraphicsItem`

`QGraphicsScene` is a surface for managing large number of graphics items. It is a container for `QGraphicsItems`. It is not a visual item, it uses the `QGraphicsView` to show items on screen. It has support for locating items and for determining the visibility of items and it is responsible for event handling and propagation. The `QGraphicsView` provides a canvas for displaying the contents of the `QGraphicsScene`. It shows graphics items in a scrollable viewport. The `QGraphicsItem` is a base class for all graphical items in a `QGraphicsScene`. It is used to create custom graphics items.

### 3.1 Simple example

First, we have a simple example covering the basics of the framework. Once we know the basics, it is quite easy to work with the framework.

---

Listing 3.1: Simple Example

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial
```

*This is a simple example of the  
Graphics View framework.*

*Author: Jan Bodnar  
Website: [zetcode.com](http://zetcode.com)  
Last edited: August 2017  
,,,*

```
from PyQt5.QtWidgets import (QApplication, QGraphicsView,  
                               QGraphicsScene)  
import sys  
  
class Example(QGraphicsView):  
  
    def __init__(self):  
        super().__init__()  
  
        self.setGeometry(300, 300, 250, 150)  
        self.setWindowTitle("Simple")  
  
        self.init()  
  
    def init(self):  
  
        self.scene = QGraphicsScene()  
        self.scene.addText("ZetCode")  
        self.setScene(self.scene)  
  
app = QApplication([])  
ex = Example()  
ex.show()  
sys.exit(app.exec_())
```

---

In the code example, we show a text in the viewport of the `QGraphicsView` widget.

```
class Example(QGraphicsView):  
  
    def __init__(self):  
        super().__init__()
```

The example inherits from the `QGraphicsView` widget. This widget will display the text.

```
self.scene = QGraphicsScene()  
self.scene.addText("ZetCode")
```

Inside the `init()` method, we create an instance of the `QGraphicsScene` class. The `addText()` method creates a text item and adds it to the scene.

```
self.setScene(self.scene)
```

The `setScene()` connects the scene with the graphics view.



Figure 3.1: Showing text in a QGraphicsView

## 3.2 Custom graphics item

In the next example, we create a custom item based on a `QGraphicsRectItem`.

Listing 3.2: Custom graphics item

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a custom
graphics item based on a QGraphicsRectItem.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QGraphicsScene,
                             QGraphicsView, QGraphicsRectItem, QGraphicsItem)
from PyQt5.QtGui import QColor, QPen
from PyQt5.QtCore import Qt
import sys

class Item(QGraphicsRectItem):

    def __init__(self, x, y, w, h):
        super().__init__(x, y, w, h)

        self.setFlag(QGraphicsItem.ItemIsMovable, True)
        self.setCursor(Qt.SizeAllCursor)
        self.setBrush(QColor(250, 100, 0))
        self.setPen(QPen(Qt.NoPen))

class Example(QGraphicsView):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 300, 300)
        self.setWindowTitle("Custom item")
```

```

        self.init()

    def init(self):

        self.scene = QGraphicsScene()

        self.item = Item(0, 0, 100, 100)
        self.scene.addItem(self.item)

        self.setScene(self.scene)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

We create our own custom item. It will be an orange rectangle. It will be movable and there will be a specific cursor shown when hovering a mouse pointer over it.

```

class Item(QGraphicsRectItem):

    def __init__(self, x, y, w, h):
        super().__init__(x, y, w, h)

```

We inherit from the built-in `QGraphicsRectItem` class. We only customize it a bit.

```

self.setFlag(QGraphicsItem.ItemIsMovable, True)

```

This flag makes the item movable on the viewport.

```

self.setCursor(Qt.SizeAllCursor)

```

This type of cursor appears when we move the mouse pointer over the area of the rectangle.

```

self.setBrush(QColor(255, 100, 0))
self.setPen(QPen(Qt.NoPen))

```

The inside of the rectangle is drawn in orange color. The outline is not drawn. The default outline would be drawn in black color.

```

self.item = Item(0, 0, 100, 100)
self.scene.addItem(self.item)

```

An instance of the custom item is created and added to the scene.

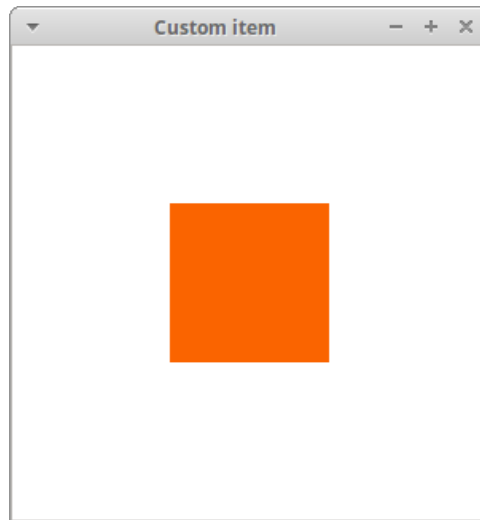


Figure 3.2: A custom graphics item

### 3.3 Rotating a rectangle

We can perform transformations on the graphics items. The next example rotates a rectangle.

Listing 3.3: Rotating a rectangle

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we rotate a rectangle.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QSlider,
                             QGraphicsRectItem, QGraphicsScene, QGraphicsItem,
                             QGraphicsView, QVBoxLayout)
from PyQt5.QtGui import QPainter, QTransform, QColor
from PyQt5.QtCore import Qt
import sys

class Rectangle(QGraphicsRectItem):

    def __init__(self, x, y, w, h):
        super().__init__(x, y, w, h)

        self.setBrush(QColor(250, 50, 0))
        self.setPen(QColor(250, 50, 0))
```



```

        self.setFlag(QGraphicsItem.ItemIsMovable, True)
        self.setCursor(Qt.SizeAllCursor)

        self.tx = 200
        self.ty = 200

    def doRotate(self, alfa):

        tr = QTransform()
        tr.translate(self.tx, self.ty)
        tr.rotate(alfa)
        tr.translate(-self.tx, -self.ty)

        self.setTransform(tr)

class View(QGraphicsView):

    def __init__(self):
        super(View, self).__init__()

        self.setRenderHint(QPainter.Antialiasing)

        self.initScene()

    def initScene(self):

        self.scene = QGraphicsScene()
        self.setSceneRect(0, 0, 400, 400)

        self.rect = Rectangle(150, 150, 100, 100)
        self.scene.addItem(self.rect)

        self.setScene(self.scene)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setWindowTitle("Rotation")
        self.setGeometry(150, 150, 300, 300)

        self.initUI()

    def initUI(self):

        vbox = QVBoxLayout()

        self.view = View()
        sld = QSlider(Qt.Horizontal, self)
        sld.setRange(-180, 180)

        sld.valueChanged[int].connect(self.changeValue)

        vbox.addWidget(self.view)
        vbox.addWidget(sld)
        self.setLayout(vbox)

```

```

def changeValue(self, value):

    self.view.rect.doRotate(value)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

An orange custom rectangle is shown in a viewport of the `QGraphicsView` widget. We have a slider widget below. We use this slider to rotate the rectangle.

```

self.tx = 200
self.ty = 200

```

These coordinates point at the middle of the rectangle.

```

def doRotate(self, alfa):

    tr = QTransform()
    tr.translate(self.tx, self.ty)
    tr.rotate(alfa)
    tr.translate(-self.tx, -self.ty)

    self.setTransform(tr)

```

The `doRotate()` performs the rotation of the rectangle. We move the origin of the transformation system to the middle of the rectangle. The rotation is performed along this origin point. We do the rotation with the `rotate()` method. Finally, we move the origin back to the previous position. Note that the transformation is done on the item. It is because items live in their own coordinate system. When creating a custom item, item coordinates are all we need to worry about.

```

self.scene = QGraphicsScene()
self.setSceneRect(0, 0, 400, 400)

```

The `setSceneRect()` defines the space of the scene. It is used by `QGraphicsView` to determine the view's default scrollable area. Sur viewable area will be 400x400 px.

```

self.view = View()
sld = QSlider(Qt.Horizontal, self)
sld.setRange(-180, 180)

```

We create an instance of the view. We create the slider widget and set a range. The values are angles for the rotation.

```

def changeValue(self, value):

    self.view.rect.doRotate(value)

```

Upon moving the slider, the `changeValue()` method is called. Here we call the `doRotate()` method of the item with a specific angle value.

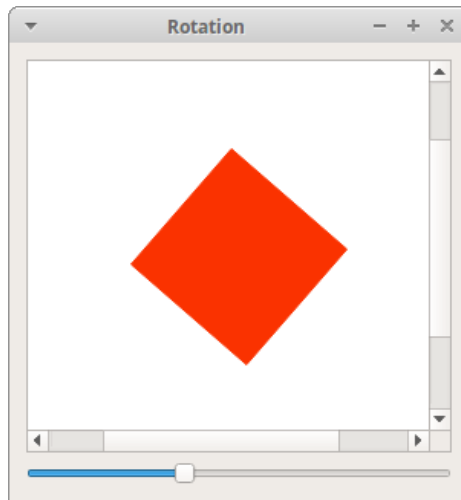


Figure 3.3: Rotation of a rectangle

### 3.4 Item animation

With the `QPropertyAnimation`, we can animate graphics items. It animates PyQt5 properties with interpolation. A class declaring properties must be a `QObject`.

Listing 3.4: Graphics View Item Animation

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This programs creates a sine wave animation of
a ball object.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QGraphicsView,
                             QGraphicsPixmapItem, QGraphicsScene, QGraphicsObject)
from PyQt5.QtGui import QPainter, QPixmap
from PyQt5.QtCore import (QObject, QRect, QPoint, QPointF,
                           QPropertyAnimation, pyqtProperty)
import sys, math

TIME = 3000

class Ball(QObject):

    def __init__(self):
        super().__init__()
```

```

        self.pixmap_item = QGraphicsPixmapItem(QPixmap("ball.png"))

    def _set_pos(self, pos):
        self.pixmap_item.setPos(pos)

    pos = pyqtProperty(QPointF, fset=_set_pos)

class MyView(QGraphicsView):

    def __init__(self):
        super().__init__()

        self.initView()

    def initView(self):

        self.ball = Ball()
        self.ball.pos = QPointF(5, 50)

        self.animation = QPropertyAnimation(self.ball, b'pos')
        self.animation.setDuration(5000);
        self.animation.setStartValue(QPointF(5, 80))

        for i in range(20):
            self.animation.setKeyValueAt(i/20,
                QPointF(i, math.sin(i))*30)

        self.animation.setEndValue(QPointF(570, 5))

        self.scene = QGraphicsScene(self)
        self.scene.setSceneRect(120, -50, 250, 150)
        self.scene.addItem(self.ball.pixmap_item)
        self.setScene(self.scene)

        self.setWindowTitle("Sine wave animation")
        self.setRenderHint(QPainter.Antialiasing)
        self.setGeometry(300, 300, 700, 200)

        self.animation.start()

app = QApplication([])
view = MyView()
view.show()
sys.exit(app.exec_())

```

---

This program creates a sine wave animation of a ball object.

```

class Ball(QObject):

    def __init__(self):
        super().__init__()

        self.pixmap_item = QGraphicsPixmapItem(QPixmap("ball.png"))

    def _set_pos(self, pos):
        self.pixmap_item.setPos(pos)

    pos = pyqtProperty(QPointF, fset=_set_pos)

```

We have a ball object. To animate an object in graphics view framework, it must inherit from `QObject` and `QGraphicsItem`. PyQt5 does not support multiple inheritance; therefore, we use composition technique. With `pyqtProperty` we create a new property.

```
self.ball = Ball()
self.ball.pos = QPointF(5, 50)
```

A ball object is created and an initial position is set.

```
self.animation = QPropertyAnimation(self.ball, b'pos')
self.animation.setDuration(5000);
self.animation.setStartValue(QPointF(5, 80))
```

`QPropertyAnimation` animates PyQt5 properties; in our case, we animate the `pos` property of the ball object. The duration of the animation is set with the `setDuration()` method. The `setStartValue()` sets the starting position of the ball.

```
for i in range(20):
    self.animation.setKeyValueAt(i/20,
        QPointF(i, math.sin(i))*30)
```

The `setKeyValueAt()` creates a key frame at the given step with the given value. We use `sin()` function to create a sine wave animation. We multiply the `sin()` value by a constant because the values of the function are too small for us.

```
self.animation.setEndValue(QPointF(570, 5))
```

With `setEndValue()` we define an ending point of the animation.

### 3.5 Collision detection

Graphics View framework supports collision detection. All the graphics items are inside the `QGraphicsScene`. The scene has a `collidingItems()` method which returns all items that collide within the scene.

Listing 3.5: Collision Detection

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This program shows collision detection with a custom
QGraphicsItem.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QGraphicsItem,
                             QGraphicsScene, QGraphicsView)
from PyQt5.QtGui import QPainter, QPainterPath, QPolygonF, QBrush
from PyQt5.QtCore import Qt, QPointF
import sys
```

```

class Item(QGraphicsItem):

    def __init__(self):
        super().__init__()

        self.brush = None
        self.createPolygon()

    def createPolygon(self):

        self.polygon = QPolygonF()
        self.polygon.append(QPointF(130, 140))
        self.polygon.append(QPointF(180, 170))
        self.polygon.append(QPointF(180, 140))
        self.polygon.append(QPointF(220, 110))
        self.polygon.append(QPointF(140, 100))

    def shape(self):

        path = QPainterPath()
        path.addPolygon(self.polygon)
        return path

    def paint(self, painter, option, widget):

        if self.brush:
            painter.setBrush(self.brush)

        painter.drawPolygon(self.polygon)

    def setBrush(self, brush):

        self.brush = brush

    def boundingRect(self):

        return self.polygon.boundingRect()

class MyView(QGraphicsView):

    def __init__(self):
        super().__init__()

        self.initView()
        self.initScene()
        self.checkCollisions()

    def initView(self):

        self.setWindowTitle("Collision detection")
        self.setRenderHint(QPainter.Antialiasing)
        self.setGeometry(300, 300, 300, 250)

```

```

def initScene(self):

    self.scene = QGraphicsScene(self)
    self.scene.setSceneRect(0, 0, 300, 250)

    self.item = Item()

    self.scene.addEllipse(160, 60, 40, 40)
    self.scene.addEllipse(80, 80, 80, 80)
    self.scene.addEllipse(190, 120, 60, 60)
    self.scene.addEllipse(150, 165, 50, 50)

    self.scene.addItem(self.item)
    self.setScene(self.scene)

def checkCollisions(self):

    items = self.scene.items()

    brush = QBrush(Qt.SolidPattern)
    brush.setStyle(Qt.HorPattern)

    for i in range(len(items)):
        item = items[i]

        if self.scene.collidingItems(item):
            item.setBrush(brush)

app = QApplication([])
view = MyView()
view.show()
sys.exit(app.exec_())

```

---

We put a polygon and four circles in the scene. Those items that collide with other items are drawn with a horizontal brush fill.

```

class Item(QGraphicsItem):

    def __init__(self):
        super().__init__()

        self.brush = None
        self.createPolygon()
...

```

The custom item is based on the basic `QGraphicsItem`.

```

def createPolygon(self):

    self.polygon = QPolygonF()
    self.polygon.append(QPointF(130, 140))
    self.polygon.append(QPointF(180, 170))
    self.polygon.append(QPointF(180, 140))
    self.polygon.append(QPointF(220, 110))
    self.polygon.append(QPointF(140, 100))

```

The `createPolygon()` method creates a custom polygon graphics item.

```

def shape(self):

```

```

path = QPainterPath()
path.addPolygon(self.polygon)
return path

```

The `shape()` method returns the painter path of the polygon.

```

def paint(self, painter, option, widget):

    if self.brush:
        painter.setBrush(self.brush)

    painter.drawPolygon(self.polygon)

```

The `paint()` method paints the item on the window. If a brush is defined, the polygon is drawn with a fill.

```

def boundingRect(self):

    return self.polygon.boundingRect()

```

The `boundingRect()` returns bounding rectangle of the graphics item.

```

self.item = Item()
self.scene.addItem(self.item)

```

An instance of the custom graphics item is created and added to the scene.

```

self.scene.addEllipse(160, 60, 40, 40)
self.scene.addEllipse(80, 80, 80, 80)
self.scene.addEllipse(190, 120, 60, 60)
self.scene.addEllipse(150, 165, 50, 50)

```

Four circles are added to the scene. Some of these objects will collide with the custom polygon item.

```

def checkCollisions(self):

    items = self.scene.items()

    brush = QBrush(Qt.SolidPattern)
    brush.setStyle(Qt.HorPattern)
    ...

```

After the scene has been set up, we call the `checkCollisions()` method. We get all the items from the scene and create a horizontal solid brush.

```

for i in range(len(items)):
    item = items[i]

    if self.scene.collidingItems(item):
        item.setBrush(brush)

```

We go through all the items. We take an item and check if it collides with another item using the `collidingItems()` method. If it does, we apply the above created brush for the item. This way all items which collide will be filled with horizontal lines.



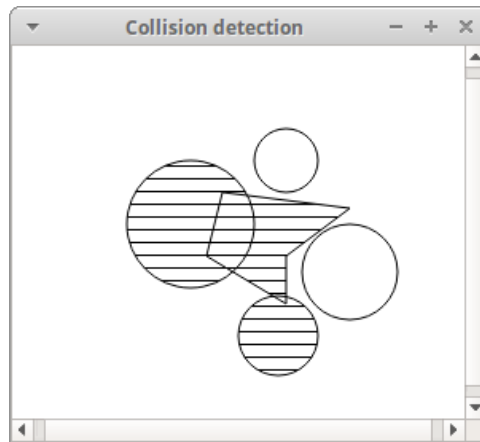


Figure 3.4: Collision detection

## 3.6 Selecting items

Graphics items can be selected with a mouse if they have the appropriate flag set. The selected items on the scene can be detected with the `selectedItems()` method.

Listing 3.6: Selecting items

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we delete
selected items from the scene.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QApplication, QWidget, QFrame,
                             QHBoxLayout, QVBoxLayout, QGraphicsItem, QGraphicsView,
                             QGraphicsScene, QPushButton )
from PyQt5.QtGui import QPainter
import sys

class View(QGraphicsView):

    def __init__(self):
        super().__init__()

        self.setRenderHint(QPainter.Antialiasing)

class Scene(QGraphicsScene):
```

```

def __init__(self):
    super().__init__()

    self.initScene()

def initScene(self):

    for i in range(5):

        e = self.addEllipse(20*i, 40*i, 50, 50)

        flag1 = QGraphicsItem.ItemIsMovable
        flag2 = QGraphicsItem.ItemIsSelectable

        e.setFlag(flag1, True)
        e.setFlag(flag2, True)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(150, 150, 350, 300)
        self.setWindowTitle("Selection")

        self.initUI()

    def initUI(self):

        hbox = QHBoxLayout()

        self.view = View()
        self.scene = Scene()
        self.view.setScene(self.scene)

        hbox.addWidget(self.view)

        frame = QFrame()

        self.delete = QPushButton("Delete", frame)
        self.delete.setEnabled(False)
        vbox = QVBoxLayout()
        vbox.addWidget(self.delete)
        vbox.addStretch(1)

        frame.setLayout(vbox)
        hbox.addWidget(frame)
        self.setLayout(hbox)

        self.delete.clicked.connect(self.onClick)
        self.scene.selectionChanged.connect(self.selChanged)

    def onClick(self):

        selectedItems = self.scene.selectedItems()

        if len(selectedItems) > 0:
            for item in selectedItems:

```

```

        self.scene.removeItem(item)

def selChanged(self):

    selectedItems = self.scene.selectedItems()

    if len(selectedItems):
        self.delete.setEnabled(True)

    else:
        self.delete.setEnabled(False)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

We have five circles on the scene. These items can be moved and selected with a mouse. We can delete selected items by clicking on the delete button. Multiple items can be selected with a control key.

```

def initScene(self):

    for i in range(5):

        e = self.addEllipse(20*i, 40*i, 50, 50)

        flag1 = QGraphicsItem.ItemIsMovable
        flag2 = QGraphicsItem.ItemIsSelectable

        e.setFlag(flag1, True)
        e.setFlag(flag2, True)

```

We create and add five circles to the scene. The `ItemIsMovable` flag makes the item movable. The `ItemIsSelectable` flag makes the item selectable.

```
self.scene.selectionChanged.connect(self.selChanged)
```

Each time an item is selected or deselected, we call the `selChanged()` method.

```

def onClick(self):

    selectedItems = self.scene.selectedItems()

    if len(selectedItems) > 0:
        for item in selectedItems:
            self.scene.removeItem(item)

```

Clicking on the delete button we delete the selected items from the scene. The `selectedItems()` method is used to determine all selected items in the scene. After that, each selected item is removed from the scene using the `removeItem()` method.

```

def selChanged(self):

    selectedItems = self.scene.selectedItems()

    if len(selectedItems):
        self.delete.setEnabled(True)

```

```

else:
    self.delete.setEnabled(False)

```

If there are any selected items, the delete button is enabled.

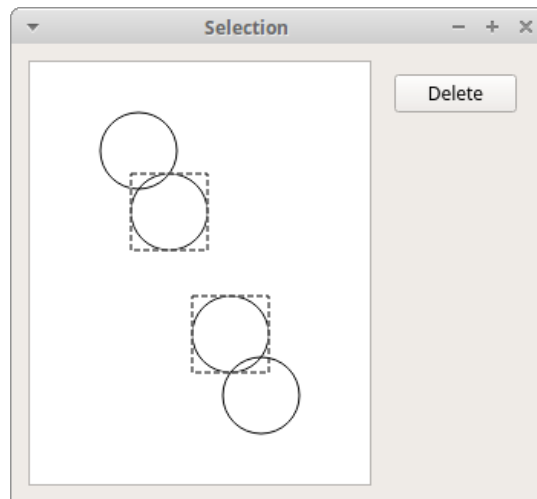


Figure 3.5: Selecting items

## 3.7 Zooming items

A common requirement in graphics programming is zooming. In this section we show how to zoom graphics items in the Graphics View Framework.

Listing 3.7: Zooming items

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we zoom
items on the QGraphicsView.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QSlider,
                             QGraphicsView, QGraphicsScene, QVBoxLayout)
from PyQt5.QtGui import QPainter, QColor
from PyQt5.QtCore import Qt
import sys

class View(QGraphicsView):

```

```

def __init__(self):
    super().__init__()

    self.setGeometry(300, 300, 300, 300)
    self.setRenderHint(QPainter.Antialiasing)

    self.init()

def init(self):

    self.scene = QGraphicsScene()

    r1 = self.scene.addRect(150, 40, 100, 100)
    r1.setBrush(QColor(250, 50, 0))
    r1.setPen(QColor(250, 50, 0))

    e1 = self.scene.addEllipse(40, 70, 80, 80)
    e1.setBrush(QColor(0, 50, 250))
    e1.setPen(QColor(0, 50, 250))

    r2 = self.scene.addRect(60, 180, 150, 70)
    r2.setBrush(QColor(0, 250, 50))
    r2.setPen(QColor(0, 250, 50))

    self.setScene(self.scene)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        vbox = QVBoxLayout()

        self.view = View()

        slider = QSlider(Qt.Horizontal, self)
        slider.setRange(1, 500)
        slider.setValue(100)
        slider.valueChanged[int].connect(self.onZoom)

        vbox.addWidget(self.view)
        vbox.addWidget(slider)

        self.setLayout(vbox)
        self.setWindowTitle("Zoom")
        self.setGeometry(150, 150, 300, 300)

    def onZoom(self, value):

        val = value / 100
        self.view.resetTransform()
        self.view.scale(val, val)

```

---

```

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

We put three objects on the scene. Below the view, we put a slider which performs the zooming on those three items.

```

def onZoom(self, value):

    val = value / 100
    self.view.resetTransform()
    self.view.scale(val, val)

```

The `onZoom()` method zooms the items on the view. The method receives a value from the slider. The value is divided by 100. This way we get a scale factor from 0.01 to 5. (The slider's range is 1..500.) The `resetTransform()` method resets the view to use the identity matrix, where one pixel in the view represents one unit on the scene. Finally, the `scale()` method performs the zooming.

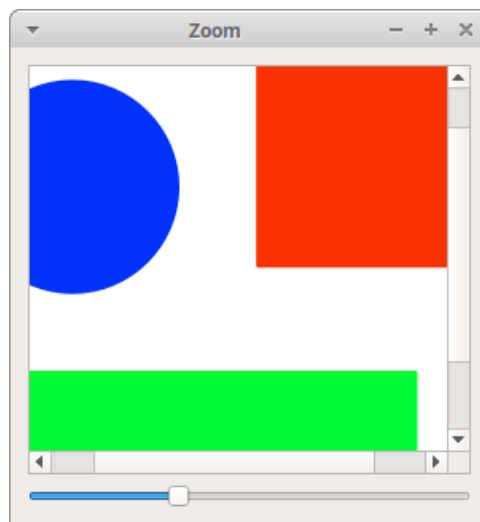


Figure 3.6: Zooming items

## 3.8 Grouping items

So far we have worked with individual objects of a scene. Graphics items can be put into groups. The `QGraphicsItemGroup` class is used to work with groups in the Graphics View Framework.

---

Listing 3.8: Grouping Items

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

```

```

'''
ZetCode Advanced PyQt5 tutorial

In this example, we group items.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QGraphicsScene,
                              QGraphicsView, QGraphicsItemGroup, QGraphicsItem,
                              QHBoxLayout)
from PyQt5.QtGui import QPainter, QColor, QPen, QBrush
from PyQt5.QtCore import Qt
import sys

class MyGroup(QGraphicsItemGroup):

    def __init__(self):
        super().__init__()

        self.setCursor(Qt.OpenHandCursor)
        self.setFlag(QGraphicsItem.ItemIsMovable)
        self.setFlag(QGraphicsItem.ItemIsSelectable, True)

    def paint(self, painter, option, widget):

        painter.setRenderHint(QPainter.Antialiasing)

        brush = QBrush(QColor("#333333"))
        pen = QPen(brush, 0.5)
        pen.setStyle(Qt.DotLine)
        painter.setPen(pen)

        if self.isSelected():
            boundRect = self.boundingRect()
            painter.drawRect(boundRect)

class Scene(QGraphicsScene):

    def __init__(self):
        super().__init__()

        self.initScene()

    def initScene(self):

        self.r1 = self.addRect(20, 50, 120, 50)
        self.r1.setFlag(QGraphicsItem.ItemIsMovable)
        self.r1.setFlag(QGraphicsItem.ItemIsSelectable, True)

        self.r2 = self.addRect(150, 100, 50, 50)
        self.r2.setFlag(QGraphicsItem.ItemIsMovable)
        self.r2.setFlag(QGraphicsItem.ItemIsSelectable, True)

        self.c = self.addEllipse(30, 150, 60, 60)
        self.c.setFlag(QGraphicsItem.ItemIsMovable)

```

```

        self.c.setFlag(QGraphicsItem.ItemIsSelectable, True)

class View(QGraphicsView):
    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 300, 300)

        policy = Qt.ScrollBarAlwaysOff

        self.setVerticalScrollBarPolicy(policy)
        self.setHorizontalScrollBarPolicy(policy)
        self.setRenderHint(QPainter.Antialiasing)
        self.setDragMode(QGraphicsView.RubberBandDrag)

        self.init()

    def init(self):

        self.group = None
        self.scene = Scene()
        self.setSceneRect(0, 0, 300, 300)
        self.setScene(self.scene)

    def keyPressEvent(self, event):

        key = event.key()

        if key == Qt.Key_U:

            if self.group != None and self.group.isSelected():

                items = self.group.childItems()
                self.scene.destroyItemGroup(self.group)
                self.group = None

                for item in items:
                    item.setSelected(False)

        if key == Qt.Key_G:

            if self.group:
                return

            selectedItems = self.scene.selectedItems()

            if len(selectedItems) > 0:
                self.group = MyGroup()

                for item in selectedItems:
                    self.group.addToGroup(item)

                self.scene.addItem(self.group)

class Example(QWidget):
    def __init__(self):

```



```

        super().__init__()

        self.initUI()

    def initUI(self):

        hbox = QHBoxLayout()

        self.view = View()
        hbox.addWidget(self.view)

        self.setLayout(hbox)
        self.setWindowTitle("Grouping")
        self.setGeometry(250, 150, 300, 300)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

There are three objects on the scene: two rectangles and a circle. We can select and move the objects. We can also select multiple objects at a time holding the control key and simultaneously clicking with a mouse. Pressing the g key, we group selected items; pressing the u key, we ungroup them. Objects in a group are selected and moved as one unit.

```

class MyGroup(QGraphicsItemGroup):

    def __init__(self):
        super().__init__()

```

We create a custom class based on the `QGraphicsItemGroup` class.

```

self.setCursor(Qt.OpenHandCursor)
self.setFlag(QGraphicsItem.ItemIsMovable)
self.setFlag(QGraphicsItem.ItemIsSelectable, True)

```

We show a hand cursor if we hover a mouse pointer over the group. We can select and move the group with a mouse.

```

def paint(self, painter, option, widget):

    painter.setRenderHint(QPainter.Antialiasing)

    brush = QBrush(QColor("#333333"))
    pen = QPen(brush, 0.5)
    pen.setStyle(Qt.DotLine)
    painter.setPen(pen)

    if self.isSelected():
        boundRect = self.boundingRect()
        painter.drawRect(boundRect)

```

If we select a group of objects, a rectangle is drawn to show the group boundaries. Dotted lines are used to draw the sides of the rectangle. The `boundingRect()` method is used to determine the bounding rectangle of the group.

```

if key == Qt.Key_U:

```

```

if self.group != None and self.group.isSelected():

    items = self.group.childItems()
    self.scene.destroyItemGroup(self.group)
    self.group = None

    for item in items:
        item.setSelected(False)

```

When the group is selected, pressing the u key will destroy it. We use a method called `destroyItemGroup()` of the scene to destroy the group. The objects of a group remain selected. So we go through all the items and deselect them. Items of a group are determined with the `childItems()` method.

```

if key == Qt.Key_G:

    if self.group:
        return

    selectedItems = self.scene.selectedItems()

    if len(selectedItems) > 0:
        self.group = MyGroup()

        for item in selectedItems:
            self.group.addToGroup(item)

    self.scene.addItem(self.group)

```

Pressing the g key, we create a group of the selected items on the scene. We use the `selectedItems()` method to figure out all the selected items of the scene. We go through the list of selected items and add them to the group using the `addToGroup()` method. The group is placed inside the scene with the `addItem()` method.

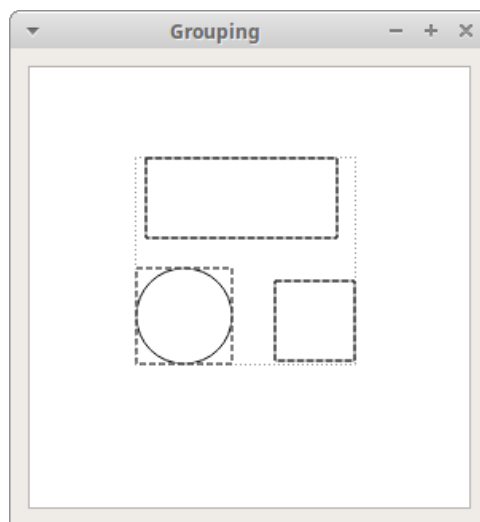


Figure 3.7: Group of items

Figure 3.7 shows all three objects placed in a group. A dotted rectangle is drawn

to show the boundaries of the group.

## 3.9 Progress meter

In this section, we create a progress meter. A progress meter is in most cases a rectangular widget which shows a progress of an event. Our progress meter will have a circular shape.

Listing 3.9: Progress Meter

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a progress meter using the
GraphicsView framework.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QGraphicsScene,
                             QGraphicsView, QGraphicsItemGroup, QGraphicsItem)
from PyQt5.QtGui import (QPainter, QColor, QPen, QBrush,
                          QRadialGradient)
from PyQt5.QtCore import (Qt, QRectF, QPointF, QTimeLine,
                           QPropertyAnimation)
import sys

class ProgressMeter(QGraphicsItem):

    def __init__(self, parent):
        super().__init__()

        self.parent = parent

        self.angle = 0
        self.per = 0

    def boundingRect(self):

        return QRectF(0, 0, self.parent.width(),
                      self.parent.height())

    def increment(self):

        self.angle += 1
        self.per = int(self.angle / 3.6)

        if self.angle > 360:
            return False
        else:
            return True
```

```

def paint(self, painter, option, widget):

    self.drawBackground(painter, widget)
    self.drawMeter(painter, widget)
    self.drawText(painter)

def drawBackground(self, painter, widget):

    painter.setRenderHint(QPainter.Antialiasing)
    painter.setPen(Qt.NoPen)

    p1 = QPointF(80, 80)
    g = QRadialGradient(p1*0.2, 80*1.1)

    g.setColorAt(0.0, widget.palette().light().color())
    g.setColorAt(1.0, widget.palette().dark().color())
    painter.setBrush(g)
    painter.drawEllipse(0, 0, 80, 80)

    p2 = QPointF(40, 40)
    g = QRadialGradient(p2, 70*1.3)

    g.setColorAt(0.0, widget.palette().midlight().color())
    g.setColorAt(1.0, widget.palette().dark().color())
    painter.setBrush(g)
    painter.drawEllipse(7.5, 7.5, 65, 65)

def drawMeter(self, painter, widget):

    painter.setPen(Qt.NoPen)
    painter.setBrush(widget.palette().highlight().color())
    painter.drawPie(7.5, 7.5, 65, 65, 0, -self.angle*16)

def drawText(self, painter):

    text = "%d%%" % self.per

    font = painter.font()
    font.setPixelSize(11)
    painter.setFont(font)
    brush = QBrush(QColor("#000000"))
    pen = QPen(brush, 1)
    painter.setPen(pen)
    painter.drawText(0, 0, 80, 80, Qt.AlignCenter, text)

class MyView(QGraphicsView):

    def __init__(self):
        super().__init__()

        self.initView()
        self.setupScene()
        self.setupAnimation()

        self.setGeometry(300, 150, 250, 250)

    def initView(self):

```

```

        self.setWindowTitle("Progress meter")
        self.setRenderHint(QPainter.Antialiasing)

        policy = Qt.ScrollBarAlwaysOff
        self.setVerticalScrollBarPolicy(policy)
        self.setHorizontalScrollBarPolicy(policy)

        self.setBackgroundBrush(self.palette().window())

        self.pm = ProgressMeter(self)
        self.pm.setPos(55, 55)

    def setupScene(self):

        self.scene = QGraphicsScene(self)
        self.scene.setSceneRect(0, 0, 250, 250)
        self.scene.addItem(self.pm)

        self.setScene(self.scene)

    def setupAnimation(self):

        self.timer = QTimeLine()
        self.timer.setLoopCount(0)
        self.timer.setFrameRange(0, 100)

        self.timer.frameChanged[int].connect(self.doStep)
        self.timer.start()

    def doStep(self, i):

        if not self.pm.increment():
            self.timer.stop()

        self.pm.update()

app = QApplication([])
view = MyView()
view.show()
sys.exit(app.exec_())

```

---

Two radial gradients are used for a background of the progress meter. They create an illusion of a three dimensional object. The progress of an event is illustrated by a pie. The pie grows until it fills the entire inner circle. The progress of an event is also marked by the percentage shown in the middle of the meter.

```

class ProgressMeter(QGraphicsItem):

    def __init__(self, parent):
        super().__init__()

```

The progress meter is based on the `QGraphicsItem` class.

```

def increment(self):

```

```

self.angle += 1
self.per = int(self.angle / 3.6)

if self.angle > 360:
    return False
else:
    return True

```

For each step, the `increment()` method is called. The method calculates the angle and the percentage.

```

def paint(self, painter, option, widget):

    self.drawBackground(painter, widget)
    self.drawMeter(painter, widget)
    self.drawText(painter)

```

The `paint()` method calls three methods. The first method draws the background of the progress meter. It uses radial gradients to create a 3D object. The second method draws the pie which is a portion of the inner circle. Finally, the third method draws the percentage of the progress of the event.

```

def drawBackground(self, painter, widget):

    painter.setRenderHint(QPainter.Antialiasing)
    painter.setPen(Qt.NoPen)

    p1 = QPointF(80, 80)
    g = QRadialGradient(p1*0.2, 80*1.1)
    ...

```

The `drawBackground()` method draws two circles with two different radial gradient fills. The `QRadialGradient` class has two parameters. The first is the center point of the gradient and the second is the radius. The center point is a tiny place over which the gradient spreads. The radius is the furthest distance of the gradient. In the above case, we create a radial gradient fill for the outer circle. It spreads from the upper left of the circle. The numbers used are a matter of a feeling rather than a precise mathematical calculation. You may want to experiment a bit with the numbers to create an effect which you like the most.

```

g.setColorAt(0.0, widget.palette().light().color())
g.setColorAt(1.0, widget.palette().dark().color())

```

A gradient is a blending of two colors used. We use colors from the widget palette for the best fit.

```

def drawMeter(self, painter, widget):

    painter.setPen(Qt.NoPen)
    painter.setBrush(widget.palette().highlight().color())
    painter.drawPie(7.5, 7.5, 65, 65, 0, -self.angle*16)

```

The `drawMeter()` method draws a pie which represents the progress of an event. The color is again chosen from the widget palette. The `drawPie()` method draws the pie. In PyQt5, the angles are specified in a 1/16 of a degree. In other words, a full circle equals to 5760 (360\*16). That is the reason, why we multiply the angle by 16. Positive values for angles mean counter clockwise direction. We want clockwise direction, so we use a negative angle for the span angle.

```
def setupAnimation(self):
    self.timer = QTimeline()
    self.timer.setLoopCount(0)
    self.timer.setFrameRange(0, 100)
    ...
```

We set up the animation for the progress meter. Setting loop count to zero will make the timeline loop forever. We stop the timeline in the `doStep()` method.

```
self.timer.frameChanged[int].connect(self.doStep)
```

Each time the frame is changed, we call the `doStep()` method.

```
self.timer.start()
```

The animation begins with the `start()` method.

```
def doStep(self, i):
    if not self.pm.increment():
        self.timer.stop()

    self.pm.update()
```

In the `doStep()` method, we call the `increment()` method of the progress meter. When we reach the full circle, the animation is terminated.

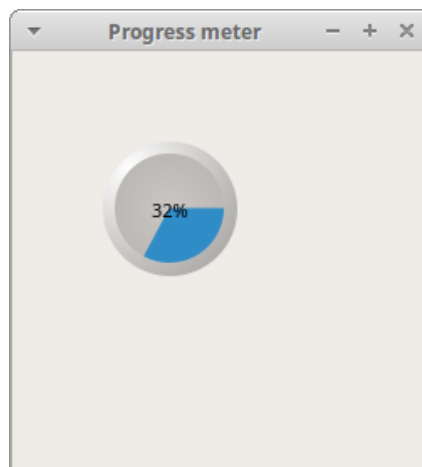


Figure 3.8: Progress meter

### 3.10 Rotating arrow example

In the next example there is an arrow in the middle of the window. The arrow always points to the mouse pointer. We face several challenges. To fully understand the example, you need to know the basics of trigonometry. It may be useful to recap the knowledge from the school.

Trigonometry is a branch of mathematics that studies triangles, particularly right triangles. Trigonometry is a Greek word which means 'measuring triangles'. Trigonometry studies triangles, angles and circles. There is a right rectangle when we move a mouse pointer on the window. The distance between the mouse pointer and the center of the arrow is the hypotenuse, which is the longest side of the right triangle. Imagine that there are two lines: one horizontal line which goes from the center of the arrow and one vertical line which goes from the mouse point. These two lines meet at a right angle. They are the legs of the right triangle.

The angle that we need for performing the rotation is calculated with arc tangent function.

```
tan(theta) = b / a
theta = atan(b / a)
```

Next imagine a circle around the arrow. As we know, a circle has 360 degrees. Rotation is absolute, theta angle is relative to the quadrant of the circle. In other words, we must calculate the theta angle differently in all four quadrants. Yet there is another thing that we must deal with. There are two measuring systems in mathematics: degrees and radians. Mathematics usually works with radians. But in PyQt5, we do rotations in degrees. We have to transform the radians into degrees.

Listing 3.10: Rotating Arrow

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we have an arrow
which always points to the mouse
pointer.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QGraphicsScene,
                             QGraphicsView, QGraphicsPixmapItem, QHBoxLayout)
from PyQt5.QtGui import QPainter, QPixmap, QTransform
from PyQt5.QtCore import Qt
import sys, math

class Arrow(QGraphicsPixmapItem):

    def __init__(self):
        super().__init__()

        self.setPixmap(QPixmap('arrow.png'))

class View(QGraphicsView):

    def __init__(self):
```



```

super(View, self).__init__()

self.setMouseTracking(True)

self.setRenderHint(QPainter.Antialiasing)

policy = Qt.ScrollBarAlwaysOff
self.setVerticalScrollBarPolicy(policy)
self.setHorizontalScrollBarPolicy(policy)

self.initScene()

def initScene(self):

    self.scene = QGraphicsScene()

    self.arrow = Arrow()
    self.scene.addItem(self.arrow)

    self.setScene(self.scene)

def mouseMoveEvent(self, event):

    s_coords = self.sceneCoordinates(event)
    theta_deg = self.calculateAngle(s_coords)

    self.doTransform(theta_deg, s_coords[2], s_coords[3])

def sceneCoordinates(self, event):

    point = event.pos()

    sMousePt = self.mapToScene(point.x(), point.y())

    s_mouse_x = sMousePt.x()
    s_mouse_y = sMousePt.y()

    arrowCenterPt = self.arrow.boundingRect().center()
    arrow_x = arrowCenterPt.x()
    arrow_y = arrowCenterPt.y()

    sArrowPt = self.arrow.mapToScene(arrow_x, arrow_y)

    s_arrow_x = sArrowPt.x()
    s_arrow_y = sArrowPt.y()

    return (s_mouse_x, s_mouse_y, arrow_x, arrow_y,
            s_arrow_x, s_arrow_y)

def calculateAngle(self, coords):

    s_mouse_x, s_mouse_y = coords[0], coords[1]
    s_arrow_x, s_arrow_y = coords[3], coords[4]

    a = abs(s_mouse_x - s_arrow_x)
    b = abs(s_mouse_y - s_arrow_y)

    if a == 0 and b == 0:

```

```

        return
    elif a == 0 and s_mouse_y < s_arrow_y:
        theta_deg = 270
    elif a == 0 and s_mouse_y > s_arrow_y:
        theta_deg = 90
    else:
        theta_rad = math.atan(b / a)
        theta_deg = math.degrees(theta_rad)

        if (s_mouse_x < s_arrow_x and \
            s_mouse_y > s_arrow_y):
            theta_deg = 180 - theta_deg

        elif (s_mouse_x < s_arrow_x and \
              s_mouse_y < s_arrow_y):
            theta_deg = 180 + theta_deg

        elif (s_mouse_x > s_arrow_x and \
              s_mouse_y < s_arrow_y):
            theta_deg = 360 - theta_deg

    return theta_deg

def doTransform(self, theta_deg, arrow_x, arrow_y):

    transform = QTransform()
    transform.translate(arrow_x, arrow_y)
    transform.rotate(theta_deg)
    transform.translate(-arrow_x, -arrow_y)

    self.arrow.setTransform(transform)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        hbox = QHBoxLayout()

        self.view = View()
        hbox.addWidget(self.view)

        self.setLayout(hbox)
        self.setWindowTitle("Arrow")
        self.setGeometry(250, 150, 390, 390)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

Hold down any mouse button and move a pointer over the area of the window. The arrow will always point at the mouse pointer.

```
class Arrow(QGraphicsPixmapItem):

    def __init__(self):
        super().__init__()

        self.setPixmap(QPixmap('arrow.png'))
```

The arrow is a `QGraphicsPixmapItem`.

```
self.setMouseTracking(True)
```

Mouse tracking is disabled by default, so the widget only receives mouse move events when at least one mouse button is pressed while the mouse is being moved. If mouse tracking is enabled, the widget receives mouse move events even if no buttons are pressed. Mouse tracking is enabled with the `setMouseTracking()`.

```
def mouseMoveEvent(self, event):

    s_coords = self.sceneCoordinates(event)
    theta_deg = self.calculateAngle(s_coords)

    self.doTransform(theta_deg, s_coords[2], s_coords[3])
```

In the `mouseMoveEvent()`, the job is divided into three methods. First, we get the scene coordinates of the center point of the arrow and the mouse pointer. Then we calculate the angle for the rotation. Finally, we perform the rotation.

```
point = event.pos()

scenePoint = self.mapToScene(point.x(), point.y())

s_mouse_x = scenePoint.x()
s_mouse_y = scenePoint.y()
```

We are in the `sceneCoordinates()` method. We get the `QPoint` for the mouse pointer. The `mapToScene()` method maps the view coordinates into scene coordinates.

```
arrowCenterPoint = self.arrow.boundingRect().center()
arrow_x = arrowCenterPoint.x()
arrow_y = arrowCenterPoint.y()

arrow = self.arrow.mapToScene(arrow_x, arrow_y)

s_arrow_x = arrow.x()
s_arrow_y = arrow.y()
```

Similarly, we get the arrow coordinates. Note that we get the center point of the arrow. We first get the bounding rectangle of the arrow and get a center point with the `center()` method of the bounding rectangle. We also map the coordinates to scene coordinates.

```
return (s_mouse_x, s_mouse_y, s_arrow_x, s_arrow_y)
```

We return the calculated scene coordinates in a tuple.

```
s_mouse_x, s_mouse_y, s_arrow_x, s_arrow_y = coords

a = abs(s_mouse_x - s_arrow_x)
b = abs(s_mouse_y - s_arrow_y)
```

Inside the `calculateAngle()` method, we first retrieve the coordinates from the `coords` tuple. We calculate the legs of the right triangle. The numbers are needed to calculate the `theta` value.

```
if a == 0 and b == 0:
    return
```

When we hover our mouse pointer over the center point of the arrow, we do no rotation.

```
elif a == 0 and s_mouse_y < s_arrow_y:
    theta_deg = 270
elif a == 0 and s_mouse_y > s_arrow_y:
    theta_deg = 90
```

We must prevent zero division error for the cases where a variable equals zero.

```
else:
    theta_rad = math.atan(b / a)
    theta_deg = math.degrees(theta_rad)
    ...
```

We calculate the arcustangent. The `math.atan()` function returns the angle value in radians. We convert the angle using the `degrees()` function of the Python `math` module.

```
if (s_mouse_x < s_arrow_x and \
    s_mouse_y > s_arrow_y):
    theta_deg = 180 - theta_deg

elif (s_mouse_x < s_arrow_x and \
    s_mouse_y < s_arrow_y):
    theta_deg = 180 + theta_deg

elif (s_mouse_x > s_arrow_x and \
    s_mouse_y < s_arrow_y):
    theta_deg = 360 - theta_deg
```

Here we map the angle for the other three quadrants. Where ever we are on the window, the `math.atan()` function returns values from 0 to 90. This is because we use a right triangle to calculate the `theta` value. The sum of the other angles always equals to 90. In order to perform correct rotation, we must consider the quadrant, in which we imagine the right triangle. Note that the quadrants go counter clockwise, while the rotation goes clockwise. The above lines adapt the angle to II., III., and IV. quadrant, respectively. This way we get values from 0 to 360.

```
transform = QTransform()
transform.translate(arrow_x, arrow_y)
transform.rotate(theta_deg)
transform.translate(-arrow_x, -arrow_y)

self.arrow.setTransform(transform)
```

These lines perform the rotation.

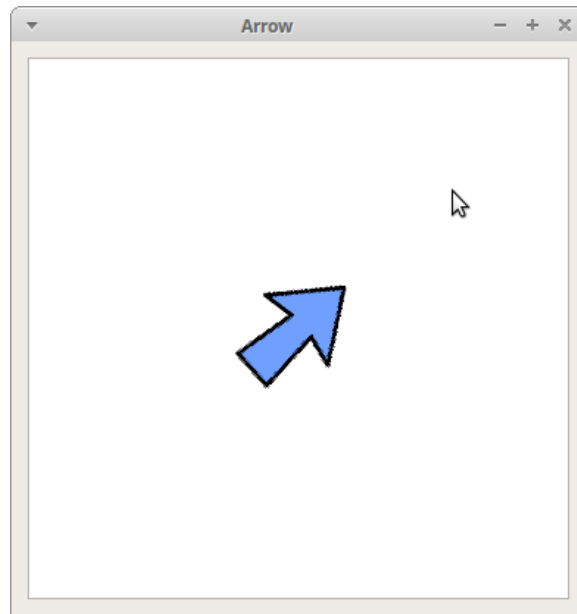


Figure 3.9: Arrow

In Figure 3.9 the arrow points north-east, where the mouse pointer was located at the moment of the screenshot.

### 3.11 Aliens

We have a small game example. There is a spacecraft and there are lots of alien ships. We navigate the spacecraft with the cursor keys. We shoot missiles with the space key and we must avoid aliens. The game is finished when we destroy all the alien ships or when we collide with an alien ship.

To create this code example, we use the Graphics View Framework. We utilize the `QBasicTimer` to set up the game cycle.

Listing 3.11: Aliens

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we we will
avoid and shoot alien ships.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QGraphicsPixmapItem,
                             QGraphicsView, QGraphicsTextItem, QGraphicsScene, QHBoxLayout)
from PyQt5.QtGui import QPainter, QPixmap, QColor
```

```

from PyQt5.QtCore import Qt, QBasicTimer
import sys

DELAY = 10
BOARD_WIDTH = 390
MISSILE_SPEED = 2
CRAFT_SIZE = 20

class Missile(QGraphicsPixmapItem):

    def __init__(self, x, y):
        super().__init__()

        self.setPixmap(QPixmap('missile.png'))

        self.x = x
        self.y = y

        self.setPos(self.x, self.y)

    def move(self):

        self.x += MISSILE_SPEED

        if (self.x > BOARD_WIDTH):
            scene = self.scene()
            scene.removeItem(self)
        else:
            self.setPos(self.x, self.y)

class Alien(QGraphicsPixmapItem):

    def __init__(self, x, y):
        super().__init__()

        self.x = x
        self.y = y

        self.setPixmap(QPixmap('alien.png'))
        self.setPos(self.x, self.y)

    def move(self):

        if self.x < 0:
            self.x = 400
        else:
            self.x = self.x - 1
            self.setPos(self.x, self.y)

class Craft(QGraphicsPixmapItem):

    def __init__(self):
        super().__init__()

        self.setPixmap(QPixmap('craft.png'))
        self.setPos(50, 50)

```

```

        self.dx = 0
        self.dy = 0

    def move(self):

        x, y = self.x(), self.y()
        x += self.dx
        y += self.dy
        self.setPos(x, y)

    def onKeyPress(self, event):

        key = event.key()

        if key == Qt.Key_Left:
            self.dx = -1

        if key == Qt.Key_Right:
            self.dx = 1

        if key == Qt.Key_Up:
            self.dy = -1

        if key == Qt.Key_Down:
            self.dy = 1

    def onKeyRelease(self, event):

        key = event.key()

        if key == Qt.Key_Left:
            self.dx = 0

        if key == Qt.Key_Right:
            self.dx = 0

        if key == Qt.Key_Up:
            self.dy = 0

        if key == Qt.Key_Down:
            self.dy = 0

class View(QGraphicsView):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 390, 390)

        self.setSceneRect(0, 0, 390, 390)
        self.setBackgroundBrush(QColor(0, 0, 0))
        self.setRenderHint(QPainter.Antialiasing)

        policy = Qt.ScrollBarAlwaysOff
        self.setVerticalScrollBarPolicy(policy)
        self.setHorizontalScrollBarPolicy(policy)

```

```

        self.init()

def init(self):

    self.pos = (
        (2380, 129), (2500, 59), (1380, 89),
        (3780, 109), (580, 139), (680, 239),
        (790, 259), (760, 150), (790, 150),
        (2930, 159), (590, 80), (530, 60),
        (1900, 259), (2660, 150), (1540, 90),
        (810, 220), (860, 320), (740, 180),
        (820, 128), (490, 170), (700, 230)
    )

    self.naliens = len(self.pos)

    self.scene = QGraphicsScene()

    self.craft = Craft()
    self.scene.addItem(self.craft)
    self.setScene(self.scene)

    for i in range(len(self.pos)):

        alien = Alien(self.pos[i][0], self.pos[i][1])
        self.scene.addItem(alien)

    text = "Aliens left: %d" % self.naliens
    self.score = self.scene.addText(text)
    self.score.setPos(15, 15)
    self.score.setDefaultTextColor(Qt.white)

    self.timer = QBasicTimer()
    self.timer.start(DELAY, self)

def timerEvent(self, event):

    self.updateScore()
    self.checkCollisions()
    self.moveItems()

def updateScore(self):

    text = "Aliens left: %d" % self.naliens
    self.score.setPlainText(text)

def fire(self):

    if not self.timer.isActive():
        return

    missile = Missile(self.craft.x() + CRAFT_SIZE+1,
        self.craft.y() + CRAFT_SIZE/2)

    self.scene.addItem(missile)

```



```

def checkCollisions(self):
    if not self.naliens:
        self.gameOver("Game won")

    items = self.items()

    for item in items:
        if isinstance(item, Missile):
            collItems = self.scene.collidingItems(item)

            if len(collItems) > 0:
                for ci in collItems:
                    self.scene.removeItem(ci)
                    self.naliens = self.naliens - 1

                self.scene.removeItem(item)

        if isinstance(item, Craft):
            collItems = self.scene.collidingItems(item)

            if item.collidesWithItem(self.score):
                continue

            if len(collItems) > 0:
                self.gameOver("Game lost")
                return

def moveItems(self):
    items = self.items()
    nItems = len(items)

    for i in range(nItems):
        item = items[i]

        if not isinstance(item, QGraphicsTextItem):
            item.move()

def gameOver(self, text):
    self.timer.stop()
    self.scene.clear()

    textItem = self.scene.addText(text)
    textItem.setDefaultTextColor(Qt.white)
    textItem.setPos(50, 50)

def keyPressEvent(self, e):
    key = e.key()

    if key == Qt.Key_Space:

```

```

        self.fire()
    else:
        self.craft.onKeyPress(e)

def keyReleaseEvent(self, e):
    self.craft.onKeyRelease(e)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        hbox = QHBoxLayout()

        self.view = View()
        hbox.addWidget(self.view)

        self.setLayout(hbox)
        self.setWindowTitle("Aliens")
        self.setGeometry(250, 150, 390, 390)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

This is a simple skeleton of a typical arcade game. We have five classes in the code example: the `Missile`, `Alien`, `Craft`, `View` and the `Example`.

The first three classes are used to build objects of the game. The `View` class is the board where the game takes place. The last one sets up the application.

```

def move(self):

    self.x += MISSILE_SPEED

    if (self.x > BOARD_WIDTH):
        scene = self.scene()
        scene.removeItem(self)
    else:
        self.setPos(self.x, self.y)

```

This is the `move()` method of the `Missile` class. It is called during each game cycle for all missiles from the scene. The method increases the `x` coordinate of the missile object. It removes the object from the scene when it goes beyond the border of the board. The `setPos()` method positions an object on the scene.

```

def move(self):

    if self.x < 0:
        self.x = 400
    else:

```

```

        self.x = self.x - 1
        self.setPos(self.x, self.y)

```

This is the `move()` method of the `Alien` class. Aliens reappear on the screen after they have disappeared on the left.

```

def move(self):

    x, y = self.x(), self.y()
    x += self.dx
    y += self.dy
    self.setPos(x, y)

```

Finally, the `move()` method of our spaceship. The `x` and `y` coordinates are increased by the `dx` and `dy` variables. These are controlled with the cursor keys.

```

def onKeyPress(self, event):

    key = event.key()

    if key == QtCore.Qt.Key_Left:
        self.dx = -1

    ...

```

If we press the left cursor key, the `dx` variable is set to -1.

```

def onKeyRelease(self, event):

    key = event.key()

    if key == QtCore.Qt.Key_Left:
        self.dx = 0

    ...

```

If we release the left cursor key, the `dx` variable is set to 0. Input and output is very time consuming. The `dx` variable continuously changes the `x` coordinate of the spaceship. We do not wait for any other key press. If one key press only increased the coordinate once, the game would be much slower.

```

self.pos = (
    (2380, 129), (2500, 59), (1380, 89),
    (3780, 109), (580, 139), (680, 239),
    ...
)

```

These are the initial coordinates of the alien ships.

```

for i in range(len(self.pos)):
    alien = Alien(self.pos[i][0], self.pos[i][1])
    self.scene.addItem(alien)

```

Aliens are created and added to the scene. Each of the objects takes one of the above `x` and `y` coordinates.

```

text = "Aliens left: %d" % self.naliens
self.score = self.scene.addText(text)
self.score.setPos(15, 15)
self.score.setDefaultTextColor(QtCore.Qt.white)

```

We put one `QGraphicsTextItem` to the scene. We will use it to show how many aliens are still left to destroy. The text item will appear in the upper left corner

of the window.

```
self.timer = QTimer()
self.timer.start(DELAY, self)
```

We set up a game cycle with the `QTimer`. Each `DELAY` milliseconds the `timerEvent()` method is called.

```
def timerEvent(self, event):

    self.updateScore()
    self.checkCollisions()
    self.moveItems()
```

The `timerEvent()` calls three distinct methods. The first method updates the score. It shows on the window how many aliens are still left. The second method checks if there are any collisions in the game. Finally, the third method moves objects (aliens, missiles and the craft) on the scene.

```
def updateScore(self):

    text = "Aliens left: %d" % self.naliens
    self.score.setPlainText(text)
```

The `naliens` variable keeps track of the number of aliens on the scene. The `setPlainText()` method sets a new text for the `QGraphicsTextItem`.

```
def fire(self):

    if not self.timer.isActive():
        return

    missile = Missile(self.craft.x() + CRAFT_SIZE+1,
                      self.craft.y() + CRAFT_SIZE/2)
    self.scene.addItem(missile)
```

Pressing the spacebar, the `fire()` method is launched. If the game cycle is finished, nothing happens. Otherwise a new missile object is created and added to the scene.

```
if not self.naliens:
    self.gameOver("Game won")
```

These are the first two lines of the `checkCollisions()` method. If there are no more aliens left, the game is won.

```
items = self.items()

for item in items:
    ...
```

We get all the items from the scene with the `items()` method. Then we iterate through the list of these items with the for loop.

```
if isinstance(item, Missile):

    colItems = self.scene.collidingItems(item)

    if len(colItems) > 0:

        for ci in colItems:
            self.scene.removeItem(ci)
```

```

        self.naliens = self.naliens - 1

        self.scene.removeItem(item)

```

In this code we check if the item is a missile object. With the `collidingItems()` we find out all items that collide with this particular missile. In our simple skeleton game the missile object can collide only with an alien. We decrease the number of aliens and remove both the aliens ship and the missile object.

```

if isinstance(item, Craft):

    colItems = self.scene.collidingItems(item)

    if item.collidesWithItem(self.score):
        continue

    if len(colItems) > 0:

        self.gameOver("Game lost")
        return

```

Now we check if our spaceship is in collision with some object. If it collides with the score text item, nothing happens. (The y coordinates of the aliens are intentionally below the text item, so this applies only to our spaceship.) If the spaceship collides with an alien, the game is over.

```

def moveItems(self):

    items = self.items()
    nItems = len(items)

    for i in range(nItems):

        item = items[i]

        if not isinstance(item, QGraphicsTextItem):
            item.move()

```

The `moveItems()` moves the objects on the scene. Except for the text item which should remain on its initial position. We iterate through all other items of the scene and call their `move()` method.

```

def gameOver(self, text):

    self.timer.stop()
    self.scene.clear()

    textItem = self.scene.addText(text)
    textItem.setDefaultTextColor(QtCore.Qt.white)
    textItem.setPos(50, 50)

```

The `gameOver()` method finishes the game. It stops the timer and clears all items from the scene with the `clear()` method. We place the final object on the scene, 'Game won' or the 'Game lost' text in white color (the background is black).

```

def keyPressEvent(self, e):

    key = e.key()

    if key == Qt.Key_Space:

```

```
        self.fire()
    else:
        self.craft.onKeyPress(e)
```

If we press a key, the `keyPressEvent()` method is called. Pressing the spacebar, the `fire()` method is launched. Otherwise, we delegate the processing to the `onKeyPress()` method of the craft object.

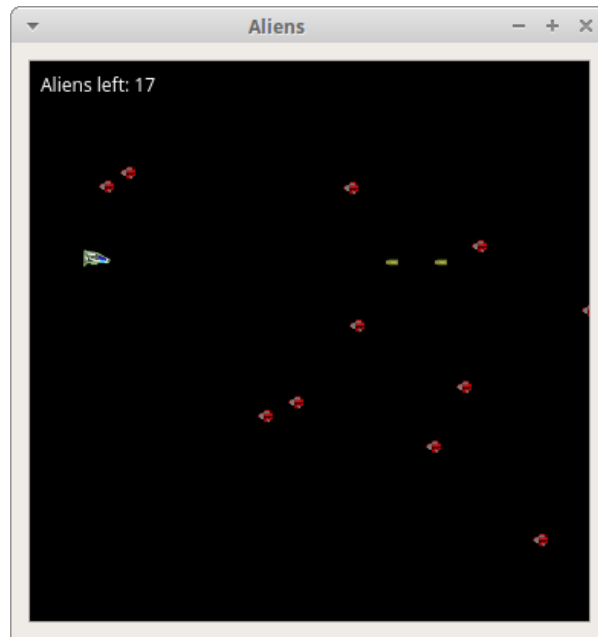


Figure 3.10: Aliens

## Chapter 4

# Layout Management

In this chapter, we cover layout management of widgets.

When we design the GUI of our application, we decide what widgets we use and how we organize those widgets in the application. To organize our widgets, we use specialized non-visible objects called layout managers.

Layout managers are software components used in widget toolkits which have the ability to lay out widgets by their relative positions without using distance units. It is often more natural to define component layouts in this manner than to define their position in pixels or common distance units, so a number of popular widget toolkits include this ability by default.

To create our layouts, we can use built-in layout managers like `QHBoxLayout`, `QVBoxLayout` or `QGridLayout` manager. We can create our own layout managers too.

### 4.1 Absolute positioning

The programmer specifies the position and the size of each component in pixels. When you use absolute positioning, you have to understand several things.

- The size and the position of a component do not change, if we resize a window.
- Applications might look different on various platforms.
- Changing fonts in your application might spoil the layout.
- If we decide to change your layout, we must completely redo the layout, which is tedious and time consuming.

In most applications, we do not use absolute positioning. However, there are some specialized areas where we can use it. For example games, specialized applications that work with diagrams, resizable components that can be moved (like a chart in a spreadsheet application), small educational examples.

In the following example, we show three images. They are placed on the window using absolute positioning. Images are loaded from the disk. They are placed inside `QLabel` widgets.

Listing 4.1: Absolute Positioning

---

```
def initUI(self):

    self.setStyleSheet("background-color: #222222")

    bardejov = QPixmap("bardejov.jpg")
    mincol = QPixmap("mincol.jpg")
    rotunda = QPixmap("rotunda.jpg")

    label1 = QLabel(self)
    label2 = QLabel(self)
    label3 = QLabel(self)

    label1.setPixmap(bardejov)
    label2.setPixmap(mincol)
    label3.setPixmap(rotunda)

    label1.setGeometry(20, 20, 120, 90)
    label2.setGeometry(40, 160, 120, 90)
    label3.setGeometry(170, 50, 120, 90)
```

---

In the code example, we show three images on the window.

```
self.setStyleSheet("background-color: #222222")
```

We set the background of the window to dark gray color.

```
bardejov = QPixmap("bardejov.jpg")
mincol = QPixmap("mincol.jpg")
rotunda = QPixmap("rotunda.jpg")
```

We create three pixmaps.

```
label1 = QLabel(self)
label2 = QLabel(self)
label3 = QLabel(self)
```

Here we create three labels.

```
label1.setPixmap(bardejov)
label2.setPixmap(mincol)
label3.setPixmap(rotunda)
```

The pixmaps are put into the labels.

```
label1.setGeometry(20, 20, 120, 90)
label2.setGeometry(40, 160, 120, 90)
label3.setGeometry(170, 50, 120, 90)
```

The `setGeometry()` method is used to position and size widgets in absolute values. The first two parameters are the `x` and `y` values of the top-left corner of the widget. The rest is the width and the height of the widget.



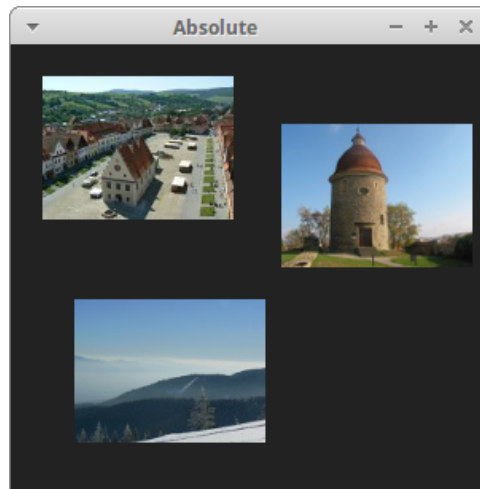


Figure 4.1: Absolute positioning

Figure 4.1 shows three images placed on the window using absolute positioning.

## 4.2 Box layout

Box layouts are very simple layout managers. They line up child widgets horizontally or vertically. The `QHBoxLayout` class lines up widgets horizontally. The `QVBoxLayout` class lines up widgets vertically. In the following example, we create a row of four buttons.

Listing 4.2: Horizontal Box Layout

---

```
def initUI(self):

    btn1 = QPushButton("Button", self)
    btn2 = QPushButton("Button", self)
    btn3 = QPushButton("Button", self)
    btn4 = QPushButton("Button", self)

    layout = QHBoxLayout()
    layout.addWidget(btn1)
    layout.addWidget(btn2)
    layout.addWidget(btn3)
    layout.addWidget(btn4)

    self.setLayout(layout)
```

---

We use a `QHBoxLayout` to create a row of four buttons.

```
layout = QHBoxLayout()
```

An instance of the `QHBoxLayout` is created.

```
layout.addWidget(btn1)
```

We add a button to the layout manager using the `addWidget()` method.

```
self.setLayout(layout)
```

We set the layout. The `QHBoxLayout` is now the layout for the parent widget.

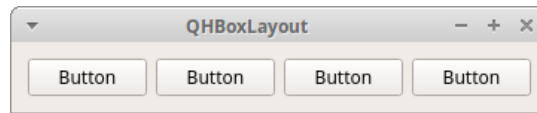


Figure 4.2: `QHBoxLayout`

Figure 4.2 shows four buttons placed in one row.

### 4.3 Size policy

`QSizePolicy` is a layout attribute which describes horizontal and vertical resizing policy. The size policy of a widget is its willingness to be resized in various ways. It affects how the widget is treated by the layout engine. Each widget returns a `QSizePolicy` that describes the horizontal and vertical resizing policy.

Widgets have their predefined size policies. These are some sensible defaults. For example a button grows horizontally but it is fixed in size vertically, when the window with a layout manager is resized.

In the following example, we show a fixed size policy.

Listing 4.3: Size Policy

---

```
def initUI(self):  
  
    btn1 = QPushButton("Button", self)  
    btn1.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)  
    btn2 = QPushButton("Button", self)  
    btn3 = QPushButton("Button", self)  
    btn3.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)  
  
    layout = QHBoxLayout()  
    layout.addWidget(btn1)  
    layout.addWidget(btn2)  
    layout.addWidget(btn3)  
  
    self.setLayout(layout)
```

---

We have three buttons. The middle button has the predefined size policy, the other two buttons have the fixed size policy defined for both directions.

```
btn1.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
```

The `setSizePolicy()` method specifies the horizontal and vertical size policies for the button widget. A widget with the `QSizePolicy.Fixed` policy does not grow or shrink in a given direction.

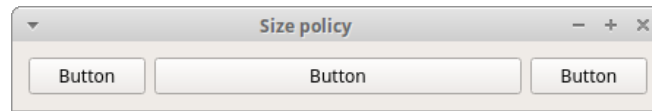


Figure 4.3: Size policy

Figure 4.3 shows the screenshot of the size policy example. The middle button grows horizontally. The other two buttons stay fixed.

## 4.4 Size hint

A size hint is a recommended size for a widget. These are the default sizes for widgets. In order to change this preferred size, we have to reimplement the `sizeHint()` method.

Listing 4.4: Size Hint

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we work with the sizeHint
property.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import QWidget, QPushButton, QApplication
from PyQt5.QtCore import QSize
import sys

class MyButton(QPushButton):

    def __init__(self, text, parent, size):
        super(MyButton, self).__init__(text, parent)

        self.size = size

    def sizeHint(self):

        return self.size

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 300, 230)
        self.setWindowTitle("Size hints")

        self.initUI()
```

```

def initUI(self):

    button1 = QPushButton("Button", self)
    button1.move(20, 50)

    button2 = MyButton("Button", self, QSize(140, 27))
    button2.move(150, 50)

    button3 = MyButton("Button", self, QSize(150, 60))

    button3.move(50, 150)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

In our example, we change a preferred size for a `QPushButton` widget.

```

def sizeHint(self):

    return self.size

```

We reimplement the `sizeHint()` method. Our method returns a size that we have provided in the constructor of the custom button.

```
button2 = MyButton("Button", self, QSize(140, 27))
```

We create an instance of a `MyButton` class. Its preferred size is set to 140x27.

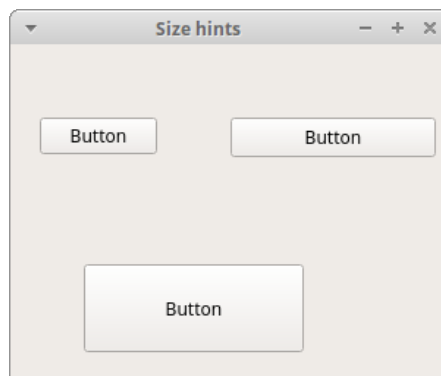


Figure 4.4: Size hint

Figure 4.4 shows three buttons. Two buttons have different default sizes than usual.

## 4.5 Minimum and maximum size

The minimum size is the widget's minimum size in pixels. The widget cannot be resized to a smaller size than its minimum widget size. The maximum size is

the widget's maximum size in pixels. The widget cannot be resized to a larger size than its maximum widget size.

Next we have a simple example demonstrating these two properties.

---

Listing 4.5: Minimum and Maximum Size

---

```
def initUI(self):

    te = QTextEdit()
    te.setMinimumSize(15, 15)
    te.setMaximumSize(350, 350)

    layout = QHBoxLayout()
    layout.addWidget(te)
    self.setLayout(layout)
```

---

In our example, we have a `QTextEdit` widget. It can be resized to a very large size by default.

```
te.setMinimumSize(15, 15)
te.setMaximumSize(350, 350)
```

Here we provide a minimum and a maximum size for the `QTextEdit` widget.

Now, when we resize the window, the widget grows up to 350x350 px and shrinks down to 15x15 px. Try to comment out these two lines and see the difference.

## 4.6 Nesting layout managers

It is possible to nest layout managers. Nesting enables us to create more complicated layouts. Box layouts are not very useful on their own. To use them practically, we have to nest them into other layout managers.

---

Listing 4.6: Nesting Layout Managers

---

```
def initUI(self):

    hbox = QHBoxLayout()

    hbox.addStretch(1)
    hbox.addWidget(QPushButton("Button"))

    vbox1 = QVBoxLayout()
    vbox1.addStretch(1)
    vbox1.addWidget(QPushButton("Button"))
    vbox1.addWidget(QPushButton("Button"))
    vbox1.addStretch(1)

    vbox2 = QVBoxLayout()
    vbox2.addStretch(1)
    vbox2.addWidget(QPushButton("Button"))
    vbox2.addWidget(QPushButton("Button"))
    vbox2.addWidget(QPushButton("Button"))
    vbox2.addStretch(1)

    vbox3 = QVBoxLayout()
    vbox3.addStretch(1)
```

```

vbox3.addWidget(QPushButton("Button"))
vbox3.addWidget(QPushButton("Button"))
vbox3.addWidget(QPushButton("Button"))
vbox3.addWidget(QPushButton("Button"))
vbox3.addStretch(1)

hbox.addLayout(vbox1)
hbox.addLayout(vbox2)
hbox.addLayout(vbox3)
hbox.addStretch(1)

self.setLayout(hbox)

```

In our example, we have one horizontal box layout. We nest three vertical boxes into this horizontal box layout. Buttons are put into these vertical boxes. We also put some stretch factors around buttons. This will push the buttons into the middle of the window. A stretch factor will be described in a more detail later.

```

hbox = QHBoxLayout()

hbox.addStretch(1)
hbox.addWidget(QPushButton("Button"))

```

We create a horizontal box layout. We add a stretch factor of 1. Everything will be pushed to the right by this stretch factor.

```

vbox1 = QVBoxLayout()
vbox1.addStretch(1)
vbox1.addWidget(QPushButton("Button"))
vbox1.addWidget(QPushButton("Button"))
vbox1.addStretch(1)

```

Here we create a vertical box layout. We put two buttons into this box. By adding a stretch factor before and after the buttons, they will be centered vertically.

```

hbox.addLayout(vbox1)
hbox.addLayout(vbox2)
hbox.addLayout(vbox3)

```

Finally, we do the nesting. We put three vertical boxes into the horizontal box. For this we use the `addLayout()` method.

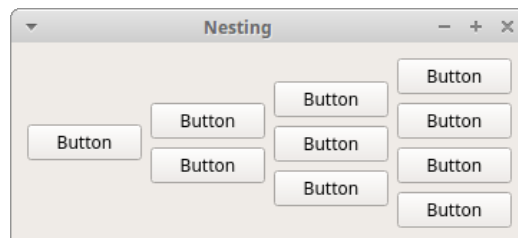


Figure 4.5: Nesting

In Figure 4.5 we have four columns of buttons. Each column is represented by a vertical box. The vertical boxes are placed in one row.

## 4.7 Stretch factor

The stretch factor is a weight that determines how widgets grow. The greater the value, the faster the growth. The stretch can be applied to widgets and to space as well.

Listing 4.7: Stretch Factor on Widgets

```
def initUI(self):  
  
    b1 = QPushButton("Button")  
    b2 = QPushButton("Button")  
    b3 = QPushButton("Button")  
  
    hbox = QHBoxLayout()  
  
    hbox.addWidget(b1)  
    hbox.addWidget(b2)  
    hbox.addWidget(b3)  
  
    hbox.setStretchFactor(b1, 1)  
    hbox.setStretchFactor(b2, 5)  
    hbox.setStretchFactor(b3, 8)  
  
    self.setLayout(hbox)
```

In our first example, we apply a stretch factor on widgets.

```
hbox.setStretchFactor(b1, 1)  
hbox.setStretchFactor(b2, 5)  
hbox.setStretchFactor(b3, 8)
```

Here we set different stretch factors on three button widgets. The greater the stretch factor, the faster the widget grows when the window is resized.

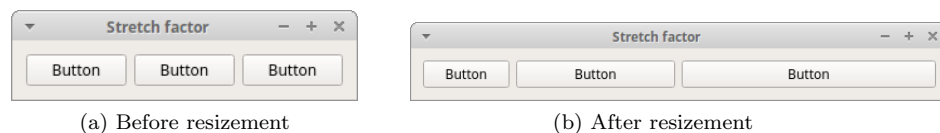


Figure 4.6: Stretch factors applied on widgets

Figure 4.6 shows the buttons with different stretch factors.

In the second example, we will apply a stretch on empty space.

Listing 4.8: Stretch Factor on Space

```
def initUI(self):

    vbox = QVBoxLayout()

    hbox1 = QHBoxLayout()
    hbox1.addStretch(1)
    hbox1.addWidget(QPushButton("Button"))
    hbox1.addWidget(QPushButton("Button"))

    hbox2 = QHBoxLayout()
    hbox2.addWidget(QPushButton("Button"))
    hbox2.addWidget(QPushButton("Button"))
    hbox2.addStretch(1)

    hbox3 = QHBoxLayout()
    hbox3.addWidget(QPushButton("Button"))
    hbox3.addStretch(1)
    hbox3.addWidget(QPushButton("Button"))

    hbox4 = QHBoxLayout()
    hbox4.addStretch(1)
    hbox4.addWidget(QPushButton("Button"))
    hbox4.addWidget(QPushButton("Button"))
    hbox4.addStretch(1)

    hbox5 = QHBoxLayout()
    hbox5.addWidget(QPushButton("Button"))
    hbox5.addStretch(1)
    hbox5.addWidget(QPushButton("Button"))
    hbox5.addStretch(1)

    vbox.addLayout(hbox1)
    vbox.addLayout(hbox2)
    vbox.addLayout(hbox3)
    vbox.addLayout(hbox4)
    vbox.addLayout(hbox5)

    self.setLayout(vbox)
```

We have one vertical box layout and we nest other five horizontal boxes into it. Each of the horizontal boxes will have two buttons. We apply the stretch factor differently on these buttons to illustrate its usage. In order to apply a stretch on the empty space, we call the `addStretch()` method.

```
hbox1 = QHBoxLayout()
hbox1.addStretch(1)
hbox1.addWidget(QPushButton("Button"))
hbox1.addWidget(QPushButton("Button"))
```

In the first row, we apply the stretch factor before the buttons. This will push the buttons to the right border.

```
hbox2 = QHBoxLayout()
hbox2.addWidget(QPushButton("Button"))
hbox2.addWidget(QPushButton("Button"))
hbox2.addStretch(1)
```



In the second row, we put a stretch factor after the two buttons. It will push them to the left border.

```
hbox3 = QHBoxLayout()
hbox3.addWidget(QPushButton("Button"))
hbox3.addStretch(1)
hbox3.addWidget(QPushButton("Button"))
```

In the third row, we put a stretch factor between the two buttons. The first button is anchored to the left, the second button to the right. There is a growable empty space between them.

```
hbox4 = QHBoxLayout()
hbox4.addStretch(1)
hbox4.addWidget(QPushButton("Button"))
hbox4.addWidget(QPushButton("Button"))
hbox4.addStretch(1)
```

In the fourth row, we put two buttons between two stretch factors. The buttons are horizontally centered.

```
hbox5 = QHBoxLayout()
hbox5.addWidget(QPushButton("Button"))
hbox5.addStretch(1)
hbox5.addWidget(QPushButton("Button"))
hbox5.addStretch(1)
```

In the final row, we add a stretch factor after each button widget. There is a growing empty space between the buttons and after the second button.

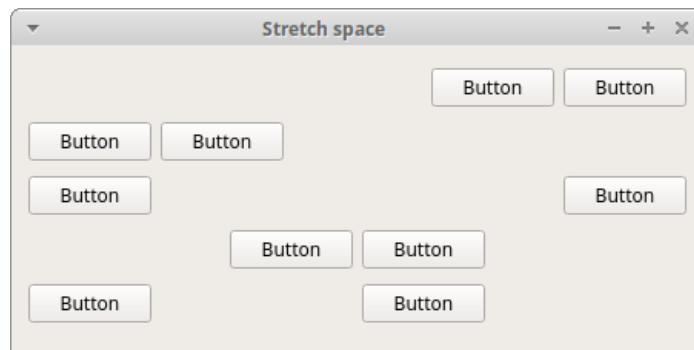


Figure 4.7: Stretching space

## 4.8 Alignment

Within layout managers, it is possible to align widgets. In a vertical box, we can align widgets to the top, center, or to the bottom. In a horizontal box, we can align widgets to the left, center, or to the right. In the previous example, we have managed to align widgets using stretch factors. We can do alignments more easily by using the convenience `setAlignment()` method.

Listing 4.9: Horizontal Alignment of Widgets

---

```
def initUI(self):

    btn1 = QPushButton("Left", self)
    btn2 = QPushButton("Center", self)
    btn3 = QPushButton("Right", self)
    btn4 = QPushButton("Stretch", self)

    vbox = QVBoxLayout()

    hbox1 = QHBoxLayout()
    hbox2 = QHBoxLayout()
    hbox3 = QHBoxLayout()

    hbox1.addWidget(btn1)
    hbox1.setAlignment(Qt.AlignLeft)
    hbox2.addWidget(btn2)
    hbox2.setAlignment(Qt.AlignCenter)
    hbox3.addWidget(btn3)
    hbox3.setAlignment(Qt.AlignRight)

    vbox.addLayout(hbox1)
    vbox.addLayout(hbox2)
    vbox.addLayout(hbox3)
    vbox.addWidget(btn4)
    self.setLayout(vbox)
```

---

In the example, we have three horizontal boxes nested in one vertical box. In these three horizontal boxes, we align button widgets to the left, center and to the right. We also add a single button to the vertical box which will stretch from left to right.

```
hbox1.addWidget(btn1)
hbox1.setAlignment(Qt.AlignLeft)
```

We add a button widget to the horizontal box. We call the `setAlignment()` method to align the button to the left. The `Qt.AlignLeft` flag is used.

```
vbox.addWidget(btn4)
```

We put a single button to the vertical box. Push buttons keep their height fixed but they can be resized horizontally. This is their default size policy. Our button will stretch from left to right.

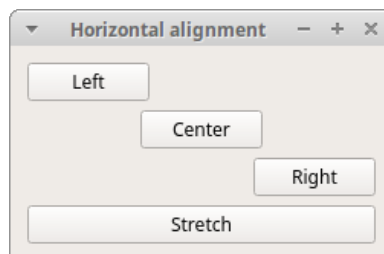


Figure 4.8: Horizontal alignment

Figure 4.8 shows horizontal alignment of button widgets.

The next example aligns widgets vertically.

Listing 4.10: Vertical Alignment of Widgets

---

```
def initUI(self):

    hbox = QHBoxLayout()

    vbox1 = QVBoxLayout()
    vbox2 = QVBoxLayout()
    vbox3 = QVBoxLayout()

    vbox1.setAlignment(Qt.AlignBottom)
    vbox1.addWidget(QPushButton("Button", self))
    vbox1.addWidget(QPushButton("Button", self))
    vbox1.addWidget(QPushButton("Button", self))

    vbox2.setAlignment(Qt.AlignCenter)
    vbox2.addWidget(QPushButton("Button", self))
    vbox2.addWidget(QPushButton("Button", self))
    vbox2.addWidget(QPushButton("Button", self))

    vbox3.setAlignment(Qt.AlignTop)
    vbox3.addWidget(QPushButton("Button", self))
    vbox3.addWidget(QPushButton("Button", self))
    vbox3.addWidget(QPushButton("Button", self))

    hbox.addLayout(vbox1)
    hbox.addLayout(vbox2)
    hbox.addLayout(vbox3)
    self.setLayout(hbox)
```

---

In the example, we have three vertical boxes nested in one horizontal box. In these three vertical boxes, we align button widgets to the top, center, and to the bottom.

```
vbox1.setAlignment(Qt.AlignBottom)
vbox1.addWidget(QPushButton("Button", self))
vbox1.addWidget(QPushButton("Button", self))
vbox1.addWidget(QPushButton("Button", self))
```

These three buttons are aligned to the bottom with the `setAlignment()` method. The `Qt.AlignBottom` flag is used.

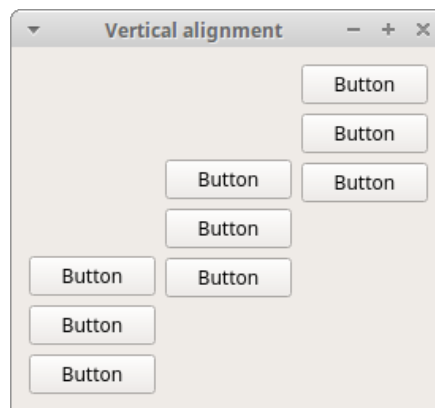


Figure 4.9: Vertical alignment

Figure 4.9 shows vertical alignment of button widgets.

## 4.9 Buttons example

Here we have our first practical example. We put two push buttons in the right-bottom corner of the window.

Listing 4.11: Buttons Example

---

```
def initUI(self):

    okBtn = QPushButton("OK")
    canBtn = QPushButton("Cancel")

    hbox = QHBoxLayout()
    hbox.addStretch(1)
    hbox.addWidget(okBtn)
    hbox.addWidget(canBtn)

    vbox = QVBoxLayout()
    vbox.addStretch(1)
    vbox.addLayout(hbox)

    self.setLayout(vbox)
```

---

To achieve this layout, we use a vertical box, a horizontal box, and stretch factors.

```
okBtn = QPushButton("OK")
canBtn = QPushButton("Cancel")
```

We create two instances of `QPushButton` widgets.

```
hbox = QHBoxLayout()
hbox.addStretch(1)
hbox.addWidget(okBtn)
hbox.addWidget(canBtn)
```

We create a horizontal layout box. We add a stretch factor to the box. Then add two push buttons to the box. The stretch factor will push the buttons to

the right of the window.

```
vbox = QVBoxLayout()
vbox.addStretch(1)
vbox.addLayout(hbox)
```

We create a vertical box layout. We add a stretch factor and nest a horizontal box into the vertical one. The stretch factor will push the horizontal box with the buttons to the bottom of the window. This way we have buttons in the right-bottom corner of the window.

```
self.setLayout(vbox)
```

We set the vertical box to be the main layout of the window.

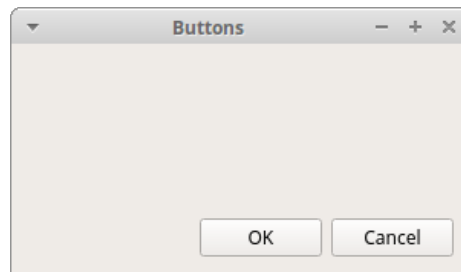


Figure 4.10: Buttons example

Figure 4.10 shows two push buttons. When the window is resized, the buttons stay in the right-bottom corner of the window.

## 4.10 Windows example

The next code example creates a Windows dialog that was found in a real world application.

Listing 4.12: Windows Example

```
def initUI(self):

    windows = QLabel('Windows')

    actBtn = QPushButton("Activate")
    clsBtn = QPushButton("Close")
    hlpBtn = QPushButton("Help")
    hlpBtn.setSizePolicy(QSizePolicy.Fixed, QSizePolicy.Fixed)
    okBtn = QPushButton("OK")

    listView = QListView()

    vbox = QVBoxLayout()
    hbox1 = QHBoxLayout()

    hbox1.addWidget(windows)
    vbox.addLayout(hbox1)

    hbox2 = QHBoxLayout()
```

```

vbox1 = QVBoxLayout()
vboxr = QVBoxLayout()

vbox1.addWidget(listView)
vboxr.addWidget(actBtn)
vboxr.addWidget(clsBtn)

vboxr.setAlignment(clsBtn, Qt.AlignTop)

hbox2.addLayout(vbox1)
hbox2.addLayout(vboxr)
vbox.addLayout(hbox2)

hbox3 = QHBoxLayout()
hbox3.addWidget(hlpBtn)
hbox3.addWidget(okBtn)
hbox3.setAlignment(okBtn, Qt.AlignRight)
vbox.addLayout(hbox3)

self.setLayout(vbox)

```

---

The Windows example is created using vertical and horizontal boxes. Together, we have three horizontal and three vertical boxes. (One vertical box is the base layout applied to the parent.) If we look at the screenshot of the example, we can see that we can divide the layout into boxes.

The Windows label goes into the first horizontal box which is nested to the base vertical box. The listview widget goes into the left vertical box. The Activate and Close buttons go into the right vertical box. These two vertical boxes are nested into the second horizontal box which is in turn nested into the base vertical box. Finally, the Help and the OK button go into the third horizontal box. It is then nested into the base vertical box as well.

```
hlpBtn.setSizePolicy(QtGui.QSizePolicy.Fixed, QtGui.QSizePolicy.Fixed)
```

The Help's button size policy set to `QSizePolicy.Fixed` in both directions. Otherwise, the button would grow horizontally.

```
vboxr.setAlignment(clsBtn, Qt.AlignTop)
```

The Close button is aligned to the top. This will push the Activate button to the top, too.

```
hbox3.setAlignment(okBtn, Qt.AlignRight)
```

The OK button is anchored to the right of the window.

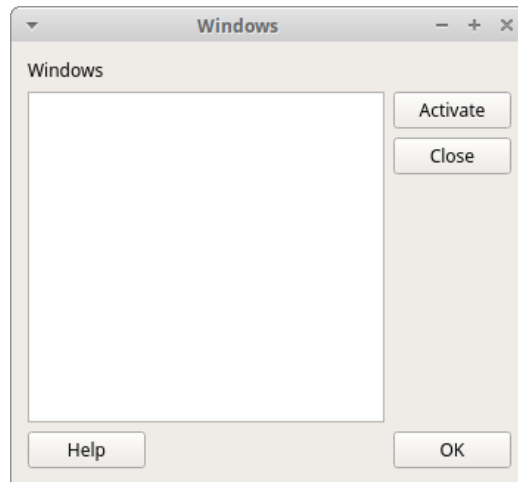


Figure 4.11: Windows example

Figure 4.11 shows the Windows example created with horizontal and vertical boxes.

## 4.11 FindReplace example

A FindReplace is another real-world dialog window which can be found in the Eclipse IDE. Here we reproduce this dialog.

Listing 4.13: FindReplace Example

---

```
def initUI(self):

    vbox = QVBoxLayout()

    hbox1 = QHBoxLayout()
    hbox1.addWidget(QLabel("Find"))
    hbox1.addWidget(QComboBox())

    vbox.addLayout(hbox1)

    hbox2 = QHBoxLayout()
    hbox2.addWidget(QLabel("Replace with"))
    hbox2.addWidget(QComboBox())

    vbox.addLayout(hbox2)

    hbox3 = QHBoxLayout()

    direction = QGroupBox("Direction")
    dirvbox = QVBoxLayout()
    dirvbox.addWidget(QCheckBox("Forward"))
    dirvbox.addWidget(QCheckBox("Backward"))
    direction.setLayout(dirvbox)

    hbox3.addWidget(direction)

    scope = QGroupBox("Scope")
```

```

scvbox = QVBoxLayout()
scvbox.addWidget(QCheckBox("All"))
scvbox.addWidget(QCheckBox("Selected items"))

scope.setLayout(scvbox)

hbox3.addWidget(scope)
vbox.addLayout(hbox3)

options = QGroupBox("Options")
opthbox = QHBoxLayout()
optvbox1 = QVBoxLayout()
optvbox2 = QVBoxLayout()

optvbox1.addWidget(QCheckBox("Case Sensitive"))
optvbox1.addWidget(QCheckBox("Whole word"))
regex = QCheckBox("Regular expressions")
optvbox1.addWidget(regex)
optvbox2.addWidget(QCheckBox("Wrap search"))
optvbox2.addWidget(QCheckBox("Incremental"))
opthbox.addLayout(optvbox1)
opthbox.addLayout(optvbox2)
options.setLayout(opthbox)

vbox.addWidget(options)

hbox4 = QHBoxLayout()
hbox4.addWidget(QPushButton("Find"))
hbox4.addWidget(QPushButton("Find/Replace"))

vbox.addLayout(hbox4)

hbox5 = QHBoxLayout()
hbox5.addWidget(QPushButton("Replace"))
hbox5.addWidget(QPushButton("Replace All"))

vbox.addLayout(hbox5)

hbox6 = QHBoxLayout()
hbox6.addStretch(1)
hbox6.addWidget(QPushButton("Close"))
vbox.addLayout(hbox6)

self.setLayout(vbox)

```

---

We have one base vertical box. The layout is divided into six horizontal boxes, which are nested into this vertical box.

```
vbox = QVBoxLayout()
```

This is the base vertical box.

```

hbox1 = QHBoxLayout()
hbox1.addWidget(QLabel("Find"))
hbox1.addWidget(QComboBox())

vbox.addLayout(hbox1)

```

First horizontal layout is created. We place a `QLabel` and a `QComboBox` into the horizontal box. The box is then nested into the base vertical box.

```
direction = QGroupBox("Direction")
```



```

dirvbox = QVBoxLayout()
dirvbox.addWidget(QCheckBox("Forward"))
dirvbox.addWidget(QCheckBox("Backward"))
direction.setLayout(dirvbox)

```

A `QGroupBox` is created. The widget provides a frame with a title and displays various other widgets. We create an instance of the `QVBoxLayout` and place two `QCheckBox` widgets into it.

```

hbox3.addWidget(direction)
...
hbox3.addWidget(scope)
vbox.addLayout(hbox3)

```

The `direction` and the `scope` group boxes are added into the third horizontal box, which is then nested into the base vertical box.

```

options = QGroupBox("Options")
opthbox = QHBoxLayout()
optvbox1 = QVBoxLayout()
optvbox2 = QVBoxLayout()

```

We have yet another `QGroupBox` widget. We call it `Options`. Its layout is a bit more complicated. It has a horizontal box and two nested vertical boxes.

```

optvbox1.addWidget(QCheckBox("Case Sensitive"))
optvbox1.addWidget(QCheckBox("Whole word"))
regex = QCheckBox("Regular expressions")
optvbox1.addWidget(regex)
optvbox2.addWidget(QCheckBox("Wrap search"))
optvbox2.addWidget(QCheckBox("Incremental"))

```

We place `QCheckBox` widgets into these vertical boxes.

```

opthbox.addLayout(optvbox1)
opthbox.addLayout(optvbox2)
options.setLayout(opthbox)

```

```

vbox.addWidget(options)

```

We nest two vertical boxes into the horizontal one. The horizontal box layout is the main layout for the `QGroupBox` widget. Finally, the `QGroupBox` widget is placed into the base vertical box.

```

hbox4 = QHBoxLayout()
hbox4.addWidget(QPushButton("Find"))
hbox4.addWidget(QPushButton("Find/Replace"))

vbox.addLayout(hbox4)

```

We create another horizontal box and add two buttons into it. The box is added into the base vertical box.

```

hbox6 = QHBoxLayout()
hbox6.addStretch(1)
hbox6.addWidget(QPushButton("Close"))
vbox.addLayout(hbox6)

```

In the end, we create our final horizontal box. We add a stretch factor and a `QPushButton` into the box. The button is right aligned. The horizontal box is nested into the base vertical layout box.

```
self.setLayout(vbox)
```

The base vertical box is set to be the main layout manager of the window.

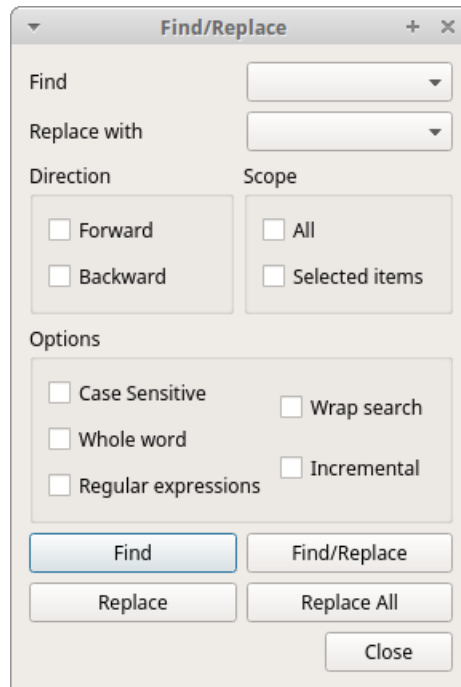


Figure 4.12: FindReplace example

Figure 4.12 shows the FindReplace example created with horizontal and vertical boxes.

## 4.12 QGridLayout manager

`QGridLayout` is the most complex layout manager. The `QGridLayout` manager places widgets in a grid of rows and columns. Intersections of rows and columns are called cells. Each widget can occupy one or more cells in the grid. A widget fills the entire cell by default. If necessary, it can fill only a portion of the cell.

### 4.12.1 Simple example

First, we have a simple example with the `QGridLayout` manager.

Listing 4.14: Simple `QGridLayout` Example

```
def initUI(self):  
    button1 = QPushButton("Button")  
    button2 = QPushButton("Button")  
    button3 = QPushButton("Button")  
    button4 = QPushButton("Button")
```

```

button5 = QPushButton("Button")
button6 = QPushButton("Button")

grid = QGridLayout()
grid.addWidget(button1, 0, 0)
grid.addWidget(button2, 0, 1)
grid.addWidget(button3, 0, 2)
grid.addWidget(button4, 1, 0)
grid.addWidget(button5, 1, 1)
grid.addWidget(button6, 1, 2)

self.setLayout(grid)

```

In this code example, we place six buttons into the `QGridLayout` manager.

```

button1 = QPushButton("Button")
button2 = QPushButton("Button")
...

```

`QPushButton` widgets are created.

```
grid = QGridLayout()
```

An instance of the `QGridLayout` manager is created.

```
grid.addWidget(button1, 0, 0)
```

We place the button at the top-left cell of the layout. The first parameter of the `addWidget()` method is the widget to be placed. The second is the row number, the third is the column number. Note that rows and columns start from zero.

```
grid.addWidget(button5, 1, 1)
```

The fifth button is placed into the second row and second column.

```
self.setLayout(grid)
```

We set the `QGridLayout` to be the main layout of the window.

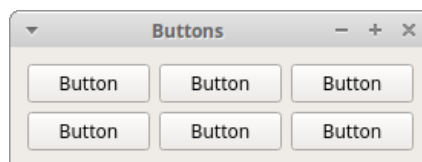


Figure 4.13: Simple `QGridLayout` example

Figure 4.13 shows a simple `QGridLayout` example. It shows a group of buttons on the window.

### 4.12.2 Spacing

We can control spacing between widgets in a `QGridLayout`. For this, we use the `setHorizontalSpacing()` method and the `setVerticalSpacing()` method.

Listing 4.15: Spacing

```
#!/usr/bin/python3
```

```

# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we set gaps
between the frame widgets.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QGridLayout,
                              QFrame, QApplication)
import sys

class MyFrame(QFrame):

    def __init__(self):
        super().__init__()

        self.setFrameStyle(QFrame.Panel | QFrame.Raised)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 350, 200)
        self.setWindowTitle("Spacing")

        self.initUI()

    def initUI(self):

        frame1 = MyFrame()
        frame2 = MyFrame()
        frame3 = MyFrame()
        frame4 = MyFrame()
        frame5 = MyFrame()
        frame6 = MyFrame()

        grid = QGridLayout()
        grid.addWidget(frame1, 0, 0)
        grid.addWidget(frame2, 0, 1)
        grid.addWidget(frame3, 0, 2)
        grid.addWidget(frame4, 1, 0)
        grid.addWidget(frame5, 1, 1)
        grid.addWidget(frame6, 1, 2)

        grid.setHorizontalSpacing(3)
        grid.setVerticalSpacing(3)

        self.setLayout(grid)

app = QApplication([])
ex = Example()
ex.show()

```

```
sys.exit(app.exec_())
```

We create six frames with some space among them. We create our own custom class which derives from the `QFrame` widget.

```
class MyFrame(QFrame):  
    def __init__(self):  
        super().__init__()  
  
        self.setFrameStyle(QFrame.Panel | QFrame.Raised)
```

This is our custom class. It inherits from the `QFrame` widget. We change the style of the frame in order to see it on the window. Otherwise, we might not be able to see it due to different widget styles.

```
grid.setHorizontalSpacing(3)
```

The `setHorizontalSpacing()` sets the horizontal spacing between widgets.

```
grid.setVerticalSpacing(3)
```

The `setVerticalSpacing()` sets the vertical spacing between widgets.

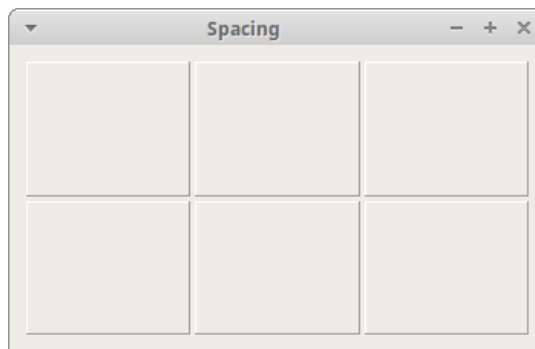


Figure 4.14: Spacing

In Figure 4.14 we see gaps among frame widgets on the window.

### 4.12.3 Spanning

Widgets can span multiple cells in a `QGridLayout`. The `addWidget()` method has parameters which control the row and the column span of a widget.

Listing 4.16: Spanning

```
def initUI(self):  
  
    grid = QGridLayout()  
    grid.addWidget(QLineEdit(), 0, 0)  
    grid.addWidget(QLineEdit(), 0, 1)  
    grid.addWidget(QLineEdit(), 0, 2)  
    grid.addWidget(QLineEdit(), 0, 3)  
  
    grid.addWidget(QLineEdit(), 1, 0, 1, 2)
```

```

grid.addWidget(QLineEdit(), 2, 0, 1, 3)
grid.addWidget(QLineEdit(), 3, 0, 1, 4)

self.setLayout(grid)

```

In this example, we use several `QLineEdit` widgets.

```

grid = QGridLayout()
grid.addWidget(QLineEdit(), 0, 0)
grid.addWidget(QLineEdit(), 0, 1)
grid.addWidget(QLineEdit(), 0, 2)
grid.addWidget(QLineEdit(), 0, 3)

```

We create an instance of the `QGridLayout` manager and place four `QLineEdit` widgets into the first row.

```
grid.addWidget(QLineEdit(), 1, 0, 1, 2)
```

Here we add another `QLineEdit` widget. The last two parameters specify the row and the column span. In our case, the widget is placed at the second row and first column. It occupies one row and spans two columns.

```
grid.addWidget(QLineEdit(), 3, 0, 1, 4)
```

This widget will span four columns.

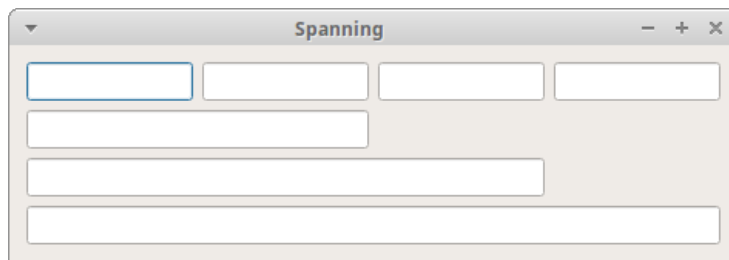


Figure 4.15: Spanning of widgets

In Figure 4.15 we see widgets that span multiple cells.

#### 4.12.4 Alignments

A widget may not fill an entire cell. It can be aligned within the cell.

Listing 4.17: Alignments

```

def initUI(self):

    button1 = QPushButton("Left")
    button2 = QPushButton("Center")
    button3 = QPushButton("Right")
    button4 = QPushButton("Stretch")

    grid = QGridLayout()
    grid.addWidget(button1, 0, 0, Qt.AlignLeft)
    grid.addWidget(button2, 1, 0, Qt.AlignCenter)
    grid.addWidget(button3, 2, 0, Qt.AlignRight)
    grid.addWidget(button4, 3, 0)

```

```
self.setLayout(grid)
```

Four buttons are placed on the window. Three buttons are aligned to the left, center and to the right. The fourth button is stretched. All four buttons are placed in one column.

```
button1 = QPushButton("Left")
button2 = QPushButton("Center")
button3 = QPushButton("Right")
button4 = QPushButton("Stretch")
```

Four push buttons are created.

```
grid.addWidget(button1, 0, 0, Qt.AlignLeft)
```

The fourth parameter of the `addWidget()` method specifies the alignment of the widget. The first button is aligned to the left.

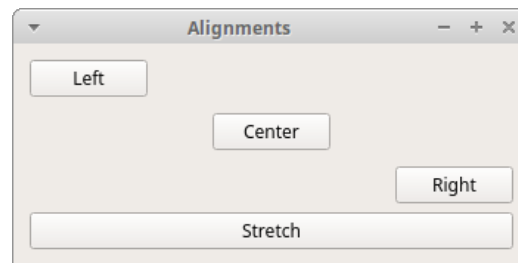


Figure 4.16: Alignments

In Figure 4.16 we see aligned widgets. Remember that all buttons are placed in one column.

#### 4.12.5 New folder example

Next we are going to create a practical example: a new folder example, created with the `QGridLayout` manager.

Listing 4.18: New Folder Example

```
def initUI(self):
    grid = QGridLayout()

    grid.addWidget(QLabel("Name: "), 0, 0)

    line = QLineEdit()
    grid.addWidget(line, 0, 1, 1, 3)

    grid.addWidget(QListView(), 1, 0, 2, 4)

    ok = QPushButton("OK")
    cancel = QPushButton("Cancel")

    grid.addWidget(ok, 3, 2)
    grid.addWidget(cancel, 3, 3)
```

```
self.setLayout(grid)
```

We have four rows and four columns in our layout.

```
grid.addWidget(QLabel("Name: "), 0, 0)
```

We place a label into the first row and first column.

```
line = QLineEdit()
grid.addWidget(line, 0, 1, 1, 3)
```

The line edit widget goes next to the label. It spans three columns.

```
grid.setColumnStretch(1, 1)
```

We make the second column take all the additional horizontal space. We get additional space when we resize the window. Two widgets occupy the second column: `QListView` and `QLineEdit` widget. They actually span through the column. When we resize the window, these widgets grow horizontally. Other widgets retain their size. If we comment out this code line, the additional space is divided among all the widgets available.

```
grid.addWidget(QListView(), 1, 0, 2, 4)
```

The `QListView` widget starts from the second row and first column. It spans two rows and four columns.

```
grid.addWidget(ok, 3, 2)
grid.addWidget(cancel, 3, 3)
```

The buttons go just below the `QListView` widget and *after* the column with the stretch factor set to 1.

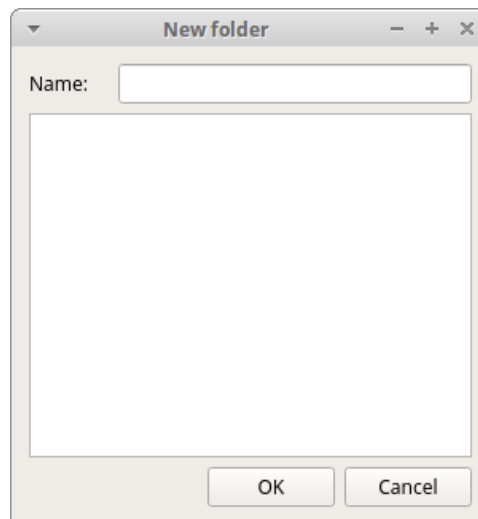


Figure 4.17: New folder

Figure 4.17 shows the Windows example created with the `QGridLayout` manager.



### 4.12.6 Windows example

This is a Windows example, created with the `QGridLayout` manager.

Listing 4.19: Windows Example

---

```
def initUI(self):

    windows = QLabel('Windows')

    actBtn = QPushButton("Activate")
    clsBtn = QPushButton("Close")
    hlpBtn = QPushButton("Help")
    okBtn = QPushButton("OK")

    listView = QListView()

    grid = QGridLayout()

    grid.addWidget(windows, 0, 0)
    grid.addWidget(listView, 1, 0, 2, 2)

    grid.addWidget(actBtn, 1, 4)
    grid.addWidget(clsBtn, 2, 4)

    grid.addWidget(hlpBtn, 4, 0)
    grid.addWidget(okBtn, 4, 4)

    grid.setColumnStretch(1, 1)
    grid.setRowStretch(2, 1)

    grid.setAlignment(clsBtn, Qt.AlignTop)

    self.setLayout(grid)
```

---

We have four buttons, a label and a listview widget.

```
grid.addWidget(windows, 0, 0)
```

The label widget is placed in the top-left cell of the layout.

```
grid.addWidget(listView, 1, 0, 2, 2)
```

The `QListView` widget goes into the second row and first column. The widget spans two rows and two columns.

```
grid.addWidget(actBtn, 1, 4)
grid.addWidget(clsBtn, 2, 4)
```

```
grid.addWidget(hlpBtn, 4, 0)
grid.addWidget(okBtn, 4, 4)
```

Buttons go into their positions. Looking at the screenshot of the example, we can figure out the cells.

```
grid.setColumnStretch(1, 1)
grid.setRowStretch(2, 1)
```

These two lines are important to understand. Try to comment them out and see the difference. All the widgets used in this example have the ability to take up additional space horizontally. This is not what we want. We want only the

`QListView` widget to grow horizontally. The default column stretch is 0. By setting the stretch factor for the second column to 1, we make the `QListView` take all the additional space horizontally.

Note that only the `QListView` occupies the second column. Vertically, the additional space is divided between the `QLabel` widget and the `QListView` widget. The default size policy does not allow a `QPushButton` widget to grow vertically. By setting the stretch factor for the third row to 1, we ensure that all additional vertical space is taken by the `QListView` widget. The label's size is fixed.

```
grid.setAlignment(clsBtn, Qt.AlignTop)
```

The Close button is aligned to the top. This will push the Activate button to the top as well.

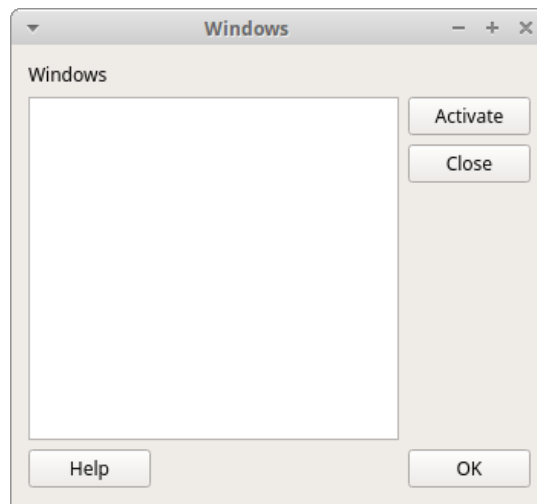


Figure 4.18: Windows

Figure 4.18 shows the Windows example created with the `QGridLayout` manager.

## Chapter 5

# Custom widgets

Toolkits usually provide only the most common widgets such as buttons, text widgets, or sliders. No toolkit can provide all possible widgets. Programmers must create other widgets by themselves. They do it by using the drawing tools provided by the toolkit. There are two possibilities: a programmer can modify or enhance an existing widget, or he can create a custom widget from scratch.

In this chapter, we are going to create three custom widgets: a led widget, a CPU widget, and a thermometer widget.

### 5.1 Led widget

A led widget is a bulb which can be set to different colors. In our case we use red, green, orange and black colors. This custom widget is simply created with an SVG image. There are four SVG images; each for one state of the led widget.

Listing 5.1: Led Widget

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a custom Led widget.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QVBoxLayout,
                             QHBoxLayout, QPushButton)
from PyQt5.QtSvg import QSvgRenderer
from PyQt5.QtGui import QPainter
from PyQt5.QtCore import Qt, QSize
import sys

class Indic:

    RED = 0
```

```

GREEN = 1
ORANGE = 2
BLACK = 3

class Led(QWidget):

    def __init__(self, parent):
        super().__init__(parent)

        self.color = Indic.GREEN
        self.setMinimumSize(QSize(50, 50))
        self.setMaximumSize(QSize(50, 50))

        self.colors = ["red.svg", "green.svg",
                       "orange.svg", "black.svg"]

    def paintEvent(self, event):

        painter = QPainter()
        painter.begin(self)
        self.drawCustomWidget(painter)
        painter.end()

    def drawCustomWidget(self, painter):

        renderer = QSvgRenderer()
        renderer.load(self.colors[self.color])
        renderer.render(painter)

    def setColor(self, newColor):

        self.color = newColor
        self.update()

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 300, 200)
        self.setWindowTitle('Led widget')

        self.initUI()

    def initUI(self):

        vbox = QVBoxLayout()
        hbox = QHBoxLayout()

        self.led = Led(self)

        hbox.addWidget(self.led)
        vbox.addStretch(1)
        vbox.addLayout(hbox)
        vbox.addStretch(1)

```

```

hbox = QHBoxLayout()

self.pb1 = QPushButton("Normal", self)
self.pb2 = QPushButton("Warning", self)
self.pb3 = QPushButton("Emergency", self)
self.pb4 = QPushButton("Off", self)

hbox.addWidget(self.pb1)
hbox.addWidget(self.pb2)
hbox.addWidget(self.pb3)
hbox.addWidget(self.pb4)

vbox.addLayout(hbox)

self.pb1.clicked.connect(self.onClick)
self.pb2.clicked.connect(self.onClick)
self.pb3.clicked.connect(self.onClick)
self.pb4.clicked.connect(self.onClick)

self.setLayout(vbox)

def onClick(self):

    sender = self.sender()
    text = sender.text()

    if text == "Normal":
        self.led.setColor(Indic.GREEN)
    elif text == "Warning":
        self.led.setColor(Indic.ORANGE)
    elif text == "Emergency":
        self.led.setColor(Indic.RED)
    elif text == "Off":
        self.led.setColor(Indic.BLACK)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

We have four buttons in a row and a led widget in the center of the window. These four buttons control the state of the led widget. There are four states for a led widget: normal, warning, emergency, and off.

```

class Indic:

    RED = 0
    GREEN = 1
    ORANGE = 2
    BLACK = 3

```

These are four values for four different states of the Led widget.

```

class Led(QWidget):

    def __init__(self, parent):
        super().__init__(parent)

```

The led widget is based on the `QWidget` object. The widget is then built on this

basic object.

```
self.color = Indic.GREEN
```

The default color is green. The `self.color` variable defines the currently selected state of the widget.

```
self.setMinimumSize(QSize(50, 50))
self.setMaximumSize(QSize(50, 50))
```

These two lines make the widget have constant size.

```
self.colors = ["red.svg", "green.svg",
               "orange.svg", "black.svg"]
```

These are the images used to create the custom widget and its various states.

```
def paintEvent(self, event):

    painter = QPainter()
    painter.begin(self)
    self.drawCustomWidget(painter)
    painter.end()
```

Inside `paintEvent()`, the led widget is drawn. The actual drawing of the widget is delegated to the `drawCustomWidget()` method.

```
def drawCustomWidget(self, painter):

    renderer = QtSvg.QSvgRenderer()
    renderer.load(self.colors[self.color])
    renderer.render(painter)
```

These lines build the custom led widget. We use the `QSvgRenderer` class to load and render an SVG image.

```
def setColor(self, newColor):

    self.color = newColor
    self.update()
```

When we click on one of the four buttons, the `setColor()` method is called. The `self.color` variable is updated and the custom widget is redrawn to reflect its new state.

```
self.led = Led(self)

hbox.addWidget(self.led)
vbox.addStretch(1)
vbox.addLayout(hbox)
vbox.addStretch(1)
```

The instance of the led widget is created. It is placed inside a horizontal layout box. Stretch factors are used to center the widget.

```
def onClick(self):

    sender = self.sender()
    text = sender.text()

    if text == "Normal":
        self.led.setColor(Indic.GREEN)
    elif text == "Warning":
```

```

        self.led.setColor(Indic.ORANGE)
    elif text == "Emergency":
        self.led.setColor(Indic.RED)
    elif text == "Off":
        self.led.setColor(Indic.BLACK)

```

The `onClick()` method is called when one of the buttons is clicked. The `sender()` method retrieves the object which has sent the signal. In our case, we find out which button it was by calling the `text()` method. We call the `setColor()` method of the led widget with an appropriate color.



Figure 5.1: Custom led widget

## 5.2 CPU widget

We call our next custom widget a CPU widget. This kind of widget is often used in system applications to measure volume, memory, or CPU usage.

Listing 5.2: CPU Widget

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a custom CPU
widget.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QSlider,
                             QApplication, QVBoxLayout)
from PyQt5.QtGui import QPainter, QColor
from PyQt5.QtCore import Qt
import sys

class CPU(QWidget):

```

```

def __init__(self, parent):
    super().__init__()

    self.parent = parent

def paintEvent(self, event):

    painter = QPainter()
    painter.begin(self)
    self.drawCustomWidget(painter)
    painter.end()

def drawCustomWidget(self, painter):

    w = self.width()
    h = self.height()

    painter.scale(w/100, h/150)
    painter.setPen(Qt.NoPen)
    painter.fillRect(0, 0, w, h, Qt.black)

    value = self.parent.getValue()
    pos = value / 5

    for i in range(1, 21):

        if i > pos:
            painter.setBrush(QColor('#075100'))
            painter.drawRect(19, 120 - i*5+1, 30, 4)
            painter.drawRect(50, 120 - i*5+1, 30, 4)

        else:
            painter.setBrush(QColor('#36ff27'))
            painter.drawRect(19, 120 - i*5+1, 30, 4)
            painter.drawRect(50, 120 - i*5+1, 30, 4)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 300, 350)
        self.setWindowTitle('CPU Widget')

        self.initUI()

    def initUI(self):

        vbox = QVBoxLayout()

        self.value = 0

        self.cpu = CPU(self)
        vbox.addWidget(self.cpu)

        self.slider = QSlider(Qt.Horizontal, self)

```



```

        self.slider.setMaximum(100)
        vbox.addWidget(self.slider)

        self.slider.valueChanged[int].connect(self.changeVal)

        self.setLayout(vbox)

    def changeVal(self):

        self.value = self.slider.value()
        self.cpu.repaint()

    def getValue(self):

        return self.value

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

We create a CPU widget and place it on the window. Below the widget, we put a `QSlider` widget which controls the CPU widget.

```

class CPU(QWidget):

    def __init__(self, parent):
        super().__init__()

        self.parent = parent

```

The CPU widget inherits from the basic `QWidget`. We keep a handle to its parent. We will use it later.

```

w = self.width()
h = self.height()

```

We determine the width and the height of the widget.

```

painter.scale(w/100, h/150)

```

This code line makes the custom widget grow as the window grows or shrink as the window shrinks. The 100x150 is a default value chosen for the widget size.

```

painter.fillRect(0, 0, w, h, Qt.black)

```

The background is filled with black color.

```

value = self.parent.getValue()
pos = value / 5

```

We get the current slider value. The slider can have values from 0 to 100. There are twenty rectangles in each of the columns. The `pos` variable tells us how many rectangles should be drawn in bright green color.

```

for i in range(1, 21):

    if i > pos:
        painter.setBrush(QColor('#075100'))

```

```

        painter.drawRect(19, 120 - i*5+1, 30, 4)
        painter.drawRect(50, 120 - i*5+1, 30, 4)

    else:
        painter.setBrush(QColor('#36ff27'))
        painter.drawRect(19, 120 - i*5+1, 30, 4)
        painter.drawRect(50, 120 - i*5+1, 30, 4)

```

Here we draw all forty rectangles, twenty in each column. We draw them either in dark or bright green color. The number of rectangles drawn in bright green color is determined by the value retrieved from the slider widget.

```

self.cpu = CPU(self)
vbox.addWidget(self.cpu)

```

The instance of the custom CPU widget is created. It is placed inside the vertical layout box.

```

self.slider = QSlider(Qt.Horizontal, self)
self.slider.setMaximum(100)

```

The QSlider widget is created. The slider's range is from 0 to 100.

```

def changeVal(self):
    self.value = self.slider.value()
    self.cpu.repaint()

```

When the slider is moved, the `changeVal()` method is called. We store the current value of the slider and repaint the CPU widget.

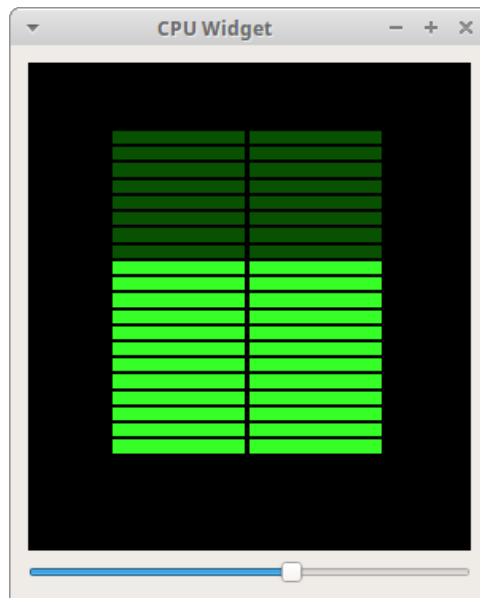


Figure 5.2: Custom CPU widget

## 5.3 Thermometer widget

Next we will create a custom Thermometer widget. The code example was adopted from a C++ example on qt-apps.org by Tomasz Ziobrowski. It is modified and simplified.

Listing 5.3: Thermometer Widget

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a custom
Thermometer widget.

Original author: Tomasz Ziobrowski
Adopted by: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication,
                              QVBoxLayout, QSlider)
from PyQt5.QtGui import (QPainter, QPainterPath, QPen, QBrush,
                          QGradient, QLinearGradient, QRadialGradient, QColor)
from PyQt5.QtCore import Qt, QRectF, QPointF
import sys

OFFSET = 10
SCALE_HEIGHT = 224

class Thermometer(QWidget):

    def __init__(self, parent):
        super().__init__(parent)

        self.value = 0

    def paintEvent(self, event):

        painter = QPainter()
        painter.begin(self)
        self.initDrawing(painter)
        self.drawTemperature(painter)
        self.drawOutline(painter)
        painter.end()

    def initDrawing(self, painter):

        self.normal = 25
        self.critical = 75
        self.m_min = 0
        self.m_max = 80

        painter.setRenderHint(QPainter.Antialiasing)
        painter.translate(self.width()/2, 0)
        painter.scale(self.height()/300, self.height()/300)
```

```

def drawOutline(self, painter):

    path = QPainterPath()

    path.moveTo(-7.5, 257)
    path.quadTo(-12.5, 263, -12.5, 267.5)
    path.quadTo(-12.5, 278, 0, 280)
    path.quadTo(12.5, 278, 12.5, 267.5)
    path.moveTo(12.5, 267.5)
    path.quadTo(12.5, 263, 7.5, 257)

    path.lineTo(7.5, 25)
    path.quadTo(7.5, 12.5, 0, 12.5)
    path.quadTo(-7.5, 12.5, -7.5, 25)
    path.lineTo(-7.5, 257)

    p1 = QPointF(-2.0, 0)
    p2 = QPointF(12.5, 0)

    linearGrad = QLinearGradient(p1, p2)
    linearGrad.setSpread(QGradient.ReflectSpread)
    linearGrad.setColorAt(1,
        QColor(0, 150, 255, 170))
    linearGrad.setColorAt(0,
        QColor(255, 255, 255, 0))

    painter.setBrush(QBrush(linearGrad))
    painter.setPen(Qt.black)
    painter.drawPath(path)

    pen = QPen()
    length = 12

    for i in range(33):

        pen.setWidthF(1.0)
        length = 12

        if i % 4:
            length = 8
            pen.setWidthF(0.8)

        if i % 2:
            length = 5
            pen.setWidthF(0.6)

        painter.setPen(pen)
        painter.drawLine(-7, 28+i*7, -7+length, 28+i*7)

    for i in range(9):

        num = self.m_min + i*(self.m_max-self.m_min)/8
        val = "{0}".format(num)
        fm = painter.fontMetrics()
        size = fm.size(Qt.TextSingleLine, val)
        point = QPointF(OFFSET,
            252-i*28+size.width()/4.0)
        painter.drawText(point, val)

def drawTemperature(self, painter):

```

```

        if self.value >= self.critical:
            color = QColor(255, 0, 0)

        elif self.value >= self.normal:
            color = QColor(0, 200, 0)

        else:
            color = QColor(0, 0, 255)

        scale = QLinearGradient(0, 0, 5, 0)
        bulb = QRadialGradient(0, 267.0, 10, -5, 262)

        scale.setSpread(QGradient.ReflectSpread)
        bulb.setSpread(QGradient.ReflectSpread)

        color.setHsv(color.hue(), color.saturation(),
                     color.value())
        scale.setColorAt(1, color)
        bulb.setColorAt(1, color)

        color.setHsv(color.hue(), color.saturation()-200,
                     color.value())
        scale.setColorAt(0, color)
        bulb.setColorAt(0, color)

        factor = self.value-self.m_min
        factor = (factor/self.m_max)-self.m_min

        temp = SCALE_HEIGHT * factor
        height = temp + OFFSET

        painter.setPen(Qt.NoPen)
        painter.setBrush(scale)
        painter.drawRect(-5, 252+OFFSET-height, 10, height)
        painter.setBrush(bulb)
        rect = QRectF(-10, 258, 20, 20)
        painter.drawEllipse(rect)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 100, 250, 350)
        self.setWindowTitle('Thermometer')

        self.initUI()

    def initUI(self):

        self.value = 0

        vbox = QVBoxLayout()

        self.the = Thermometer(self)
        vbox.addWidget(self.the)

        self.slider = QSlider(Qt.Horizontal, self)
        vbox.addWidget(self.slider)

```

```

        self.slider.setMaximum(80)

        self.slider.valueChanged[int].connect(self.changeVal)

        self.setLayout(vbox)

    def changeVal(self):

        self.the.value = self.slider.value()
        self.the.repaint()

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

There are two widgets on the window. A custom thermometer and a `QSlider` widget. The slider widget controls the temperature of the thermometer.

```

self.initDrawing(painter)
self.drawTemperature(painter)
self.drawOutline(painter)

```

Inside the `paintEvent()` method, there are three user defined methods. The `initDrawing()` method does some preliminary work. The `drawTemperature()` method draws the inside of the custom widget. The `drawOutline()` method draws the outside of the custom widget.

```

self.normal = 25
self.critical = 75
self.m_min = 0
self.m_max = 80

```

Inside the `initDrawing()` method, we do some initializations. We draw the temperature in blue color for values from 0 to `self.normal`. From `self.normal` to `self.critical` we draw in green color. Temperatures higher than `self.critical` are drawn in red color. The `self.m_min` and `self.m_max` are the minimal and maximal values on our scale.

```

painter.translate(self.width()/2, 0)
painter.scale(self.height()/300, self.height()/300)

```

We translate the drawing into the middle of the window. The `scale()` method makes the custom widget grow or shrink when the whole window grows or shrinks.

```

path = QPainterPath()

```

We use the `QPainterPath` to draw the outlines of the thermometer widget. It is a container for painting operations, enabling more complicated graphical shapes to be constructed and reused.

```

path.moveTo(-7.5, 257)
path.quadTo(-12.5, 263, -12.5, 267.5)
path.quadTo(-12.5, 278, 0, 280)
path.quadTo(12.5, 278, 12.5, 267.5)
path.moveTo(12.5, 267.5)
path.quadTo(12.5, 263, 7.5, 257)

```

These lines draw the bulb - the bottom part of the object. We use the `quadTo()` method to draw Bézier curves.

```
path.lineTo(7.5, 25)
path.quadTo(7.5, 12.5, 0, 12.5)
path.quadTo(-7.5, 12.5, -7.5, 25)
path.lineTo(-7.5, 257)
```

These code lines draw the upper part of the object. We draw both straight lines and Bézier curves.

```
...
linearGrad = QLinearGradient(p1, p2)
linearGrad.setSpread(QGradient.ReflectSpread)
...
```

We use a `QLinearGradient` to fill the scale with some light blue color. The reflected spread makes it look better.

```
for i in range(33):
    ...
```

Now we are going to draw the scale. There are 33 lines in the scale.

```
pen.setWidthF(1.0)
length = 12

if i % 4:
    length = 8
    pen.setWidthF(0.8)

if i % 2:
    length = 5
    pen.setWidthF(0.6)
```

Here we decide how long and how thick the line on the scale will be.

```
for i in range(9):

    num = self.m_min + i*(self.m_max-self.m_min)/8
    val = "{0}".format(num)
    fm = painter.fontMetrics()
    size = fm.size(Qt.TextSingleLine, val)
    point = QPointF(OFFSET, 252-i*28+size.width()/4.0)
    painter.drawText(point, val)
```

Next to the lines, we draw numbers: 0, 10, 20, ..., 80. We get the font metrics to get the size of the font.

```
if self.value >= self.critical:
    color = QColor(255, 0, 0)

elif self.value >= self.normal:
    color = QColor(0, 200, 0)

else:
    color = QColor(0, 0, 255)
```

Here we decide what kind of color we use to draw the temperature. It is based on the current value from the slider.

```
scale = QLinearGradient(0, 0, 5, 0)
bulb = QRadialGradient(0, 267.0, 10, -5, 262)
```

The drawing of the temperature has two parts: the bulb and the scale. We use a `QLinearGradient` for the scale and `QRadialGradient` for the bulb.

```
color.setHsv(color.hue(), color.saturation()-200,  
             color.value())
```

The color for the second stop point has a decreased saturation. This creates a very nice effect.

```
factor = self.value-self.m_min  
factor = (factor/self.m_max)-self.m_min  
  
temp = SCALE_HEIGHT * factor  
height = temp + OFFSET
```

We compute the height of the temperature. The actual value in range 0..80 has to be transformed to fit the height of the scale. The `OFFSET` is the distance between the first line of the scale and the ellipse in the bulb.

```
painter.setPen(Qt.NoPen)  
painter.setBrush(scale)  
painter.drawRect(-5, 252+OFFSET-height, 10, height)  
painter.setBrush(bulb)  
rect = QRectF(-10, 258, 20, 20)  
painter.drawEllipse(rect)
```

These lines draw the temperature in the scale and bulb.

```
self.the = Thermometer(self)  
vbox.addWidget(self.the)
```

The instance of the thermometer widget is created. It is placed inside the vertical box.

```
self.slider = QSlider(Qt.Horizontal, self)  
vbox.addWidget(self.slider)  
self.slider.setMaximum(80)
```

`QSlider` widget is created and placed below the custom thermometer widget. The slider's range is 0..80.

```
def changeVal(self):  
  
    self.the.value = self.slider.value()  
    self.the.repaint()
```

When we move the slider, the `changeVal()` method is called. We retrieve the slider's value and update the thermometer's `value`. The custom widget is repainted to reflect the new value.



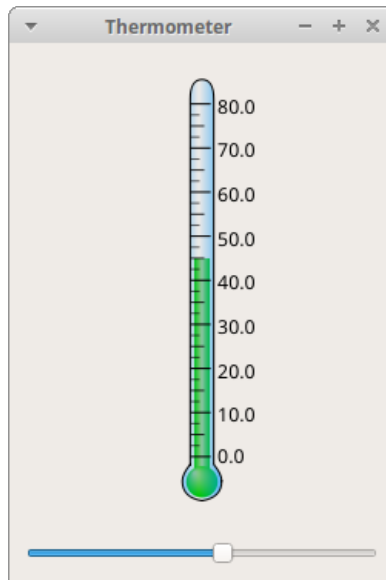


Figure 5.3: Custom Thermometer widget

## Chapter 6

# Model and View widgets

PyQt5 has several widgets which are based on the Model and View programming paradigm. This programming paradigm is an architectural pattern where the data is separated from the user interface. The data is called the model and the user interface the view. Using the Model and View pattern makes the code more flexible. Changes to the user interface will not affect data handling. And the data can be reorganized without changing the user interface.

In PyQt5 we also have another object called the delegate. The delegate is responsible for rendering the items of data.

In this chapter, we are going to work with three Model and View widgets: `QListView`, `QTreeView` and `QTableView`.

### 6.1 Basic examples

First, we are going to create some basic Model and View examples.

#### 6.1.1 QTreeView basic example

Our first code example is a simple demonstration of the Model and View programming. We display the local file system in the `QTreeView` widget.

---

Listing 6.1: Basic QTreeView Example

---

```
def initUI(self):

    self.model = QFileSystemModel()
    homedir = QDir.home().path()
    self.model.setRootPath(homedir)

    tv = QTreeView(self)
    tv.setModel(self.model)

    layout = QVBoxLayout()
    layout.addWidget(tv)
    self.setLayout(layout)
```

---

In the program, we use two classes: `QFileSystemModel` and `QTreeView`. The `QFileSystemModel` class provides a data model for the local filesystem. We use this model to display local filesystem in the `QTreeView`.

```
self.model = QFileSystemModel()
```

We create a model.

```
homedir = os.getenv('HOME')
```

Here we get the home directory.

```
self.model.setRootPath(homedir)
```

We install a file watcher on the model with `setRootPath()`. Any changes to files and directories within this directory will be reflected in the model. Note that if we do not call this method, no data is displayed in the view.

```
tv = QTreeView(self)
tv.setModel(self.model)
```

A `QTreeView` is created. The `setModel()` method attaches the model to the view.

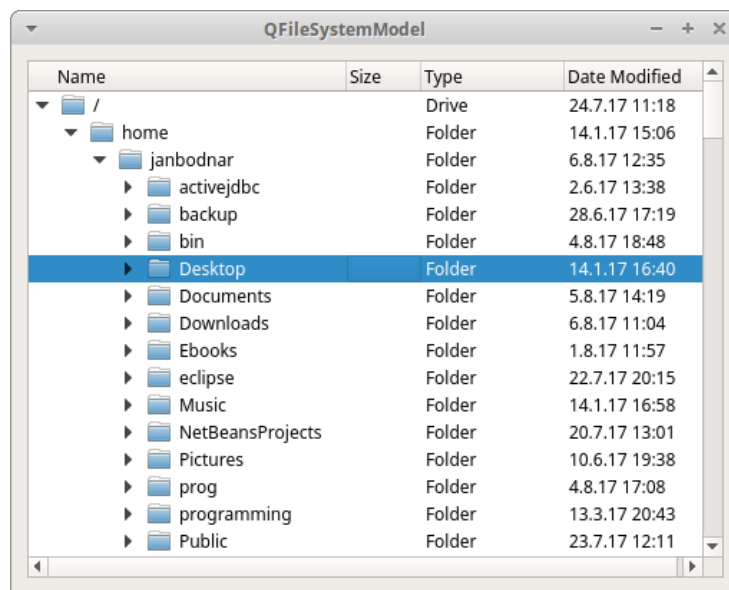


Figure 6.1: Simple Model and View example

### 6.1.2 QListView basic example

In the following example we use the `QStringListModel` to display string data in the `QListView` widget.

Listing 6.2: QListView Basic Example

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
```

```

'''
ZetCode Advanced PyQt5 tutorial

This is a basic QListView example.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication,
                             QListView, QVBoxLayout)
from PyQt5.QtCore import QStringListModel
import sys

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 350, 300)
        self.setWindowTitle("QListView")

        self.initData()
        self.initUI()

    def initData(self):

        names = ["Jack", "Tom", "Lucy", "Bill", "Jane"]
        self.model = QStringListModel(names)

    def initUI(self):

        lv = QListView(self)
        lv.setModel(self.model)

        layout = QVBoxLayout()
        layout.addWidget(lv)
        self.setLayout(layout)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

In the above example, we display string data in the `QListView` widget.

```

self.initData()
self.initUI()

```

The initialization of the model and the view is separated into two steps.

```

def initData(self):

    names = ["Jack", "Tom", "Lucy", "Bill", "Jane"]
    self.model = QStringListModel(names)

```

Inside the `initData()` method, we create a list of strings. The list is set to the

```
QStringListModel.  
  
lv = QListView(self)  
lv.setModel(self.model)
```

During the view creation, we create a `QListView` widget and attach a model to it with `setModel()`.

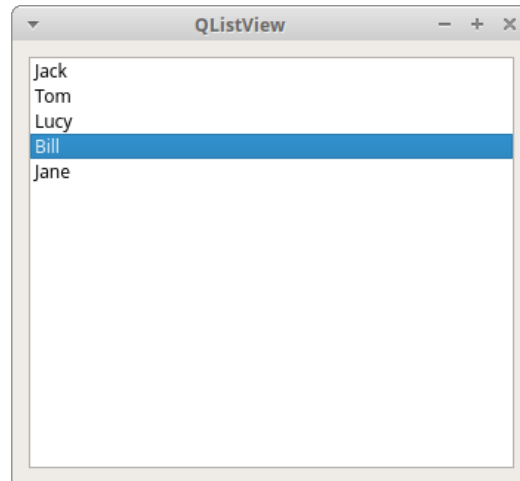


Figure 6.2: Basic `QListView` example

### 6.1.3 `QTableView` basic example

The third example displays string values in the `QTableView` widget.

Listing 6.3: `QTableView` Basic Example

---

```
#!/usr/bin/python3  
# -*- coding: utf-8 -*-  
  
'''  
ZetCode Advanced PyQt5 tutorial  
  
This is a basic QTableView example.  
  
Author: Jan Bodnar  
Website: zetcode.com  
Last edited: August 2017  
'''  
  
from PyQt5.QtWidgets import (QWidget, QApplication, QTableView,  
                               QVBoxLayout)  
from PyQt5.QtGui import QStandardItemModel, QStandardItem  
import sys  
  
class Example(QWidget):  
    def __init__(self):
```

```

        super().__init__()

        self.setGeometry(300, 300, 400, 300)
        self.setWindowTitle("QTableView")

        self.initData()
        self.initUI()

    def initData(self):

        data = ("blue", "green", "yellow", "red")
        self.model = QStandardItemModel(10, 6)

        row = 0
        col = 0

        for i in data:

            item = QStandardItem(i)
            self.model.setItem(row, col, item)
            row = row + 1

    def initUI(self):

        self.tv = QTableView(self)
        self.tv.setModel(self.model)

        vbox = QVBoxLayout()
        vbox.addWidget(self.tv)
        self.setLayout(vbox)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

In this example, we use the `QStandardItemModel` to display data in the `QTableView` widget. The `QStandardItemModel` class provides a generic model for storing custom data.

```

self.initData()
self.initUI()

```

First we initiate the data model, then we build the user interface.

```
data = ("blue", "green", "yellow", "red")
```

This is the data to be displayed.

```
self.model = QStandardItemModel(10, 6)
```

We create a new item model that has initially ten rows and six columns.

```

for i in data:

    item = QStandardItem(i)
    self.model.setItem(row, col, item)
    row = row + 1

```

We iterate through the data and set it to the model. The `QStandardItem` provides an item of the `QStandardItemModel` class. We do not work with raw data.

```
self.tv = QTableView(self)
self.tv.setModel(self.model)
```

Inside the `initUI()` method, we create the `QTableView` widget and attach the created model to it.

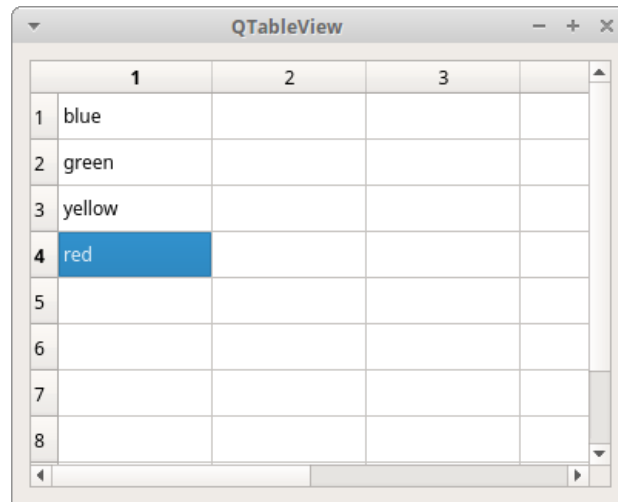


Figure 6.3: Basic `QTableView` example

## 6.2 Sorting

If the model and the view enables it, we can use the built-in sorting. In other cases, we use a proxy model to transform the structure of the model before presenting the data in the view.

### 6.2.1 Built-in sorting

The first example demonstrates the built-in sorting of the `QTableView` widget.

Listing 6.4: Built-in Sorting

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we sort data in the
QTableView widget.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
```

```

'''
from PyQt5.QtWidgets import (QWidget, QApplication, QTableView,
                              QPushButton, QHBoxLayout, QVBoxLayout,
                              QCheckBox, QSizePolicy)
from PyQt5.QtCore import Qt, QStringListModel
import sys

FIRST_COLUMN = 0

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 350, 250)
        self.setWindowTitle("Sorting")

        self.initData()
        self.initUI()

    def initData(self):

        names = ["Jack", "Tom", "Lucy", "Bill", "Jane"]
        self.model = QStringListModel(names)

    def initUI(self):

        self.tv = QTableView(self)
        self.tv.setModel(self.model)
        self.tv.setSortingEnabled(True)

        sortBtn = QPushButton("Sort", self)
        sortBtn.setSizePolicy(QSizePolicy.Fixed,
                              QSizePolicy.Fixed)

        self.sortType = QCheckBox("Ascending", self)
        self.sortType.setSizePolicy(QSizePolicy.Fixed,
                                    QSizePolicy.Fixed)

        sortBtn.clicked.connect(self.sortItems)

        hbox = QHBoxLayout()
        hbox.addWidget(self.sortType)
        hbox.addWidget(sortBtn)

        vbox = QVBoxLayout()
        vbox.addWidget(self.tv)
        vbox.addLayout(hbox)
        self.setLayout(vbox)

    def sortItems(self):

        checked = self.sortType.isChecked()

        if checked:
            self.tv.sortByColumn(FIRST_COLUMN,
                                Qt.AscendingOrder)

```



```

        else:
            self.tv.sortByColumn(FIRST_COLUMN,
                                Qt.DescendingOrder)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

In the above example, we use the `QStringListModel` for our data and use the `QTableView` to display the data. It is possible to sort data in two ways: either by clicking on the column header or by clicking on the sort button. The method invoked by the signal from the button will sort the data programatically.

```

names = ["Jack", "Tom", "Lucy", "Bill", "Jane"]
self.model = QStringListModel(names)

```

We initiate the data model.

```

self.tv = QTableView(self)
self.tv.setModel(self.model)

```

We create a view and bind the model to the view.

```

self.tv.setSortingEnabled(True)

```

The `setSortingEnabled()` method enables sorting. A small icon is displayed in the column header. By clicking on the column header, we can sort data in ascending or descending order.

```

sortBtn = QPushButton("Sort", self)
sortBtn.setSizePolicy(QSizePolicy.Fixed,
                     QSizePolicy.Fixed)

self.sortType = QCheckBox("Ascending", self)

```

We create a `QPushButton` and a `QCheckBox`. We place them below the `QTableView` widget. These two widgets sort the data programatically.

```

def sortItems(self):

    checked = self.sortType.isChecked()

    if checked:
        self.tv.sortByColumn(FIRST_COLUMN,
                              Qt.AscendingOrder)

    else:
        self.tv.sortByColumn(FIRST_COLUMN,
                              Qt.DescendingOrder)

```

The push button invokes the `sortItems()` method. We figure out the sorting order and call the `sortByColumn()` method with the chosen sorting order.

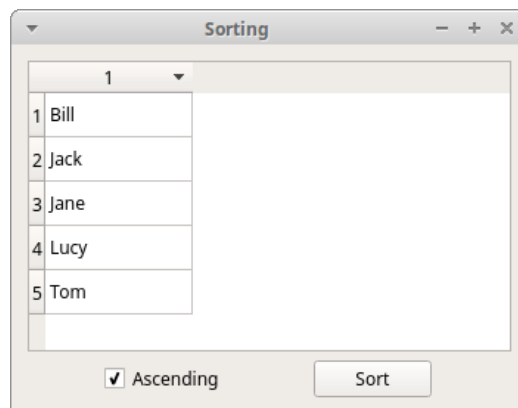


Figure 6.4: Built-in sorting

## 6.2.2 Sorting with proxy models

The second code example demonstrates sorting of data with the proxy models.

Listing 6.5: Sorting with Proxy Models

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we sort items in
a QListView using a QSortFilterProxyModel class.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QListView,
                              QPushButton, QCheckBox)
from PyQt5.QtCore import (Qt, QStringListModel,
                           QSortFilterProxyModel)
import sys

FIRST_COLUMN = 0

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 350, 200)
        self.setWindowTitle("Sorting")

        self.initData()
        self.initUI()

    def initData(self):
```

```

names = ["Jack", "Tom", "Lucy", "Bill", "Jane"]
self.model = QStringListModel(names)

self.model = QStringListModel(names)
self.filterModel = QSortFilterProxyModel(self)
self.filterModel.setSourceModel(self.model)

def initUI(self):

    sortB = QPushButton("Sort", self)
    sortB.move(250, 20)

    self.sortType = QCheckBox("Ascending", self)
    self.sortType.move(250, 60)

    sortB.clicked.connect(self.sortItems)

    self.lv = QListView(self)
    self.lv.setModel(self.filterModel)
    self.lv.setGeometry(20, 20, 200, 150)

def sortItems(self):

    checked = self.sortType.isChecked()

    if checked:
        self.filterModel.sort(FIRST_COLUMN,
                               Qt.AscendingOrder)

    else:
        self.filterModel.sort(FIRST_COLUMN,
                               Qt.DescendingOrder)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

We have a `QStringListModel` and a `QListView` class. To sort the data, we use the `QSortFilterProxyModel`.

```

self.model = QStringListModel(names)
self.filterModel = QSortFilterProxyModel(self)
self.filterModel.setSourceModel(self.model)

```

We create a `QStringListModel` from a list of strings. Then we create an instance of the `QSortFilterProxyModel` class. We attach the string list model to the filter proxy model by calling the `setSourceModel()` method.

```

def sortItems(self):

    checked = self.sortType.isChecked()

    if checked:
        self.filterModel.sort(FIRST_COLUMN,
                               Qt.AscendingOrder)

```

```

else:
    self.filterModel.sort(FIRST_COLUMN,
                          Qt.DescendingOrder)

```

We call the `sort()` method on the filter model either in ascending or in descending order.

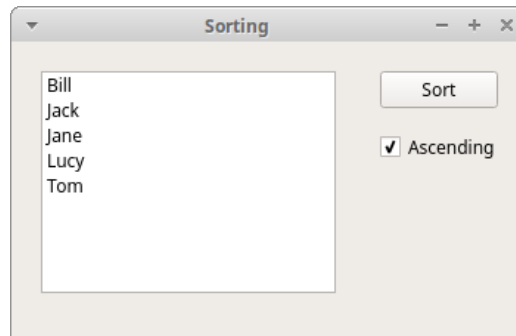


Figure 6.5: Sorting using proxy model

## 6.3 QModelIndex

`QModelIndex` class is used to locate data in a data model. The index is used by item views, delegates, and selection models to locate an item in the model.

In our example, we display data from a `QStandardItemModel` in a `QTreeView`. We use the `QModelIndex` to retrieve data from the model.

Listing 6.6: Model Index

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we work with QModelIndex
and QTreeView.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QTreeView,
                             QVBoxLayout, QAbstractItemView, QLabel)
from PyQt5.QtGui import QStandardItemModel, QStandardItem
import sys

data = ( ["Jessica Alba", "Pomona", "1981"],
         ["Angelina Jolie", "New York", "1975"],
         ["Natalie Portman", "Yerusalem", "1981"],

```

```

        ["Scarlett Johansson", "New York", "1984"] )

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 350, 250)
        self.setWindowTitle("Actresses")

        self.initData()
        self.initUI()

    def initData(self):

        self.model = QStandardItemModel()
        labels = ("Name", "Place", "Born")
        self.model.setHorizontalHeaderLabels(labels)

        for i in range(len(data)):

            name = QStandardItem(data[i][0])
            place = QStandardItem(data[i][1])
            born = QStandardItem(data[i][2])
            self.model.appendRow((name, place, born))

    def initUI(self):

        tv = QTreeView(self)
        tv.setRootIsDecorated(False)
        tv.setModel(self.model)
        behavior = QAbstractItemView.SelectRows
        tv.setSelectionBehavior(behavior)

        self.label = QLabel(self)

        layout = QVBoxLayout()
        layout.addWidget(tv)
        layout.addWidget(self.label)
        self.setLayout(layout)

        tv.clicked.connect(self.onClicked)

    def onClicked(self, idx):

        row = idx.row()
        cols = self.model.columnCount()

        data = []

        for col in range(cols):

            item = self.model.item(row, col)
            data.append(item.text())

        self.label.setText(", ".join(data))

```

```

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

We have actresses in a `QTreeView`. When we click on a specific row of the view, the data from the row is displayed in a label which is located below the view. To retrieve the data, we use the `QModelIndex` class.

```

data = ( ["Jessica Alba", "Pomona", "1981"],
          ["Angelina Jolie", "New York", "1975"],
          ["Natalie Portman", "Yerusalem", "1981"],
          ["Scarlett Jonahsson", "New York", "1984"] )

```

This is the data to be displayed.

```

self.model = QStandardItemModel()

```

To store our data, we use the `QStandartItemModel`.

```

labels = ("Name", "Place", "Born")
self.model.setHorizontalHeaderLabels(labels)

```

The `setHorizontalHeaderLabels()` method sets the horizontal header labels using a Python tuple.

```

for i in range(len(data)):
    name = QStandardItem(data[i][0])
    place = QStandardItem(data[i][1])
    born = QStandardItem(data[i][2])
    self.model.appendRow((name, place, born))

```

In this loop, we fill the model from the data tuple. We cannot append raw data directly to the model. We must first create a `QStandardItem` for each data item. The items are added to the model with `appendRow()`.

```

tv = QTreeView(self)
tv.setRootIsDecorated(False)

```

We create the `QTreeView` widget. We disable the root item of the widget with `setRootIsDecorated()`, since we do not need it.

```

tv.setModel(self.model)

```

The model is attached to the view.

```

behavior = QAbstractItemView.SelectRows
tv.setSelectionBehavior(behavior)

```

A view can have various selection behaviors. Clicking on an item, a row, a column or a single item can be selected. In our case, a row is selected.

```

tv.clicked.connect(self.onClicked)

```

By clicking on the tree view, we invoke the `onClicked()` method. The method obtains a `QModelIndex` as a parameter.

```

row = idx.row()

```

From the model index we get the row on which we have clicked with `row()`.

```
cols = self.model.columnCount()
```

We get the number of columns from the model with `columnCount()`.

```
data = []
```

```
for col in range(cols):
```

```
    item = self.model.item(row, col)
    data.append(item.text())
```

We iterate through all the columns of the row. We get items from all cells of the row in question.

```
self.label.setText(", ".join(data))
```

The final string is shown in the label widget.

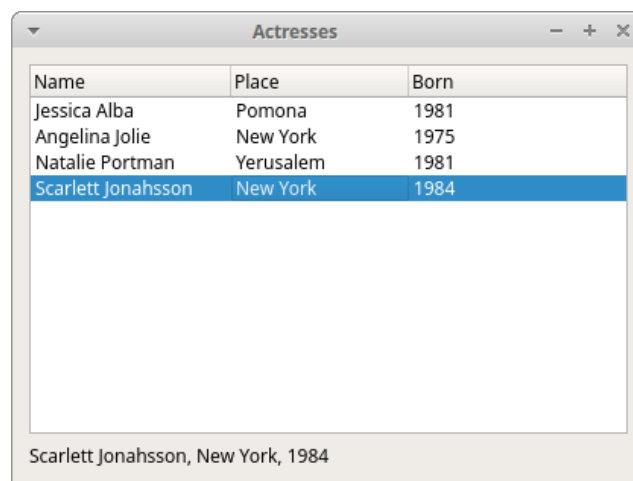


Figure 6.6: QModelIndex with QTreeView

## 6.4 QItemSelectionModel

`QItemSelectionModel` keeps track of a view's selected items. It also keeps track of the currently selected item in a view.

Listing 6.7: Item Selection Model

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we calculate
the sum of numbers in the QTableView

Author: Jan Bodnar
```

Website: [zetcode.com](http://zetcode.com)  
Last edited: August 2017  
,,

```
from PyQt5.QtWidgets import (QMainWindow, QApplication,
                              QTableView, QAction)
from PyQt5.QtGui import QStandardItemModel, QStandardItem, QIcon
import sys

class Example(QMainWindow):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 450, 350)
        self.setWindowTitle("Selection")

        self.initData()
        self.initUI()

    def initUI(self):

        self.tv = QTableView(self)
        self.tv.setModel(self.model)
        icon = QIcon('sum.png')
        self.nsum = QAction(icon, 'Sum', self)
        self.nsum.setShortcut('Ctrl+Q')

        self.nsum.triggered.connect(self.onClicked)

        self.toolbar = self.addToolBar('Sum')
        self.toolbar.addAction(self.nsum)

        self.setCentralWidget(self.tv)

    def onClicked(self):

        nsum = 0

        selmod = self.tv.selectionModel()
        selection = selmod.selection()

        if not selection.count():
            return

        indexes = selection.indexes()

        for idx in indexes:

            d = idx.data()

            if d:
                try:
                    num = int(d)
                except ValueError:
                    num = 0
            else:
                num = 0

            nsum += num
```



```

        lastIndex = indexes[-1]
        r, c = lastIndex.row(), lastIndex.column()

        item = QStandardItem(str((nsum)))
        self.model.setItem(r+1, c, item)

    def initData(self):

        self.model = QStandardItemModel(15, 15)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

We have a `QTableView` widget. We write some numbers into the cells. There is a sum button on a toolbar that computes the sum of all selected numbers.

```
nsum = 0
```

This is the variable used to calculate the sum of all selected numbers.

```
selmod = self.tv.selectionModel()
```

The `selectionModel()` retrieves the selection model from the `QTableView`.

```
selection = selmod.selection()
```

We get selected ranges of items from the view with `selection()`.

```
if not selection.count():
    return
```

We do nothing if there is no selection.

```
indexes = selection.indexes()
```

Now we get the indexes of the selected items with `indexes()`.

```

for idx in indexes:

    d = idx.data()

    if d:
        try:
            num = int(d)
        except ValueError:
            num = 0
    else:
        num = 0

    nsum += num

```

We iterate through the indexes and get the values. We calculate the sum.

```

lastIndex = indexes[-1]
r, c = lastIndex.row(), lastIndex.column()

```

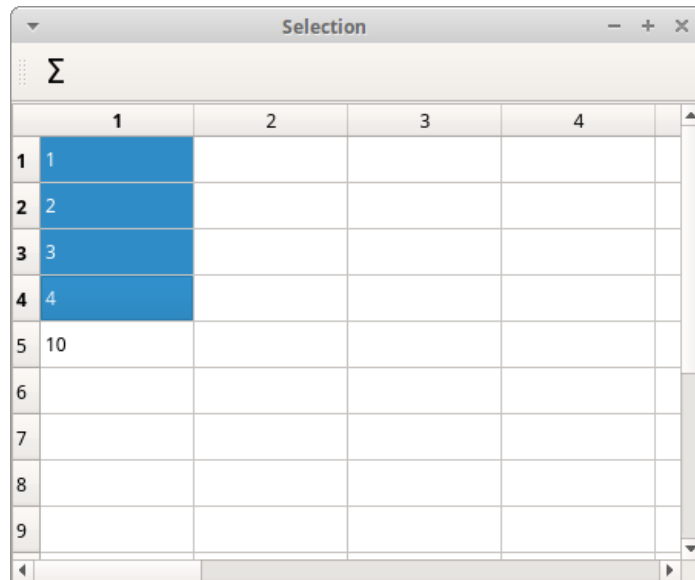
We get the row and column number of the last item in our selection.

```

item = QStandardItem(str((nsum)))
self.model.setItem(r+1, c, item)

```

We go one row below that last row and put the sum into that cell utilizing the `setItem()` method.



|    | 1  | 2 | 3 | 4 |  |
|----|----|---|---|---|--|
| 1  | 1  |   |   |   |  |
| 2  | 2  |   |   |   |  |
| 3  | 3  |   |   |   |  |
| 4  | 4  |   |   |   |  |
| 5  | 10 |   |   |   |  |
| 6  |    |   |   |   |  |
| 7  |    |   |   |   |  |
| 8  |    |   |   |   |  |
| 9  |    |   |   |   |  |
| 10 |    |   |   |   |  |

Figure 6.7: Computing the sum of values

## 6.5 Subclassing models

PyQt5 provides the following predefined models:

- `QStringListModel`
- `QStandardItemModel`
- `QFileSystemModel`
- `QSqlQueryModel`
- `QSqlTableModel`
- `QSqlRelationalTableModel`

It is possible to create our custom implementations of models if the predefined models do not meet our needs. We can subclass `QAbstractItemModel`, `QAbstractListModel`, or `QAbstractTableModel` to create your own custom models.

In our first example, we subclass the `QAbstractListModel`. We provide a read-only access to the data.

## Listing 6.8: Subclassing QAbstractListModel

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we create a read-only
QAbstractListModel subclass.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication,
                              QListView, QVBoxLayout)
from PyQt5.QtCore import (Qt, QAbstractListModel,
                           QModelIndex, QVariant)
import sys

class MyListModel(QAbstractListModel):

    def __init__(self, lang):
        super().__init__()

        self.lang = lang

    def data(self, index, role):

        if index.isValid() and role == Qt.DisplayRole:
            return QVariant(self.lang[index.row()])

        else:
            return QVariant()

    def rowCount(self, index=QModelIndex()):

        return len(self.lang)

    def flags(self, index):

        fg1 = Qt.ItemIsEnabled
        fg2 = Qt.ItemIsSelectable

        return fg1 | fg2

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 250, 200)
        self.setWindowTitle("Subclassing")

        self.initData()
```

```

        self.initUI()

    def initData(self):

        languages = ("Python", "Ruby", "Java",
                    "C", "C#", "C++")

        self.model = MyListModel(languages)

    def initUI(self):

        lv = QListView(self)
        lv.setModel(self.model)

        layout = QVBoxLayout()
        layout.addWidget(lv)
        self.setLayout(layout)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

This simple example only shows a list of programming languages in a `QListView` widget. The example uses its own model based on a `QAbstractListModel` class. Our custom model must implement the following three methods:

- `data()`
- `rowCount()`
- `flags()`

The `data()` method is used to supply data to the view. The `rowCount()` method tells how many rows the model provides. And the `flags()` method informs whether an item in the model is selectable, editable, enabled etc.

```

class MyListModel(QAbstractListModel):

    def __init__(self, lang):
        super().__init__()

        self.lang = lang

```

Our custom `MyListModel` is derived from a `QAbstractListModel`. The custom model takes a list of languages as a parameter.

```

def data(self, index, role):

    if index.isValid() and role == Qt.DisplayRole:
        return QVariant(self.lang[index.row()])

    else:
        return QVariant()

```

The `data()` method provides the data to the view. Each data item can have multiple roles associated with it. The most common role is the display role. It

controls the data to be rendered in the form of text. For this role, we send data from the `self.lang` list. Note that we must send the data encapsulated in the `QVariant` object.

```
def rowCount(self, index=QModelIndex()):  
  
    return len(self.lang)
```

The `rowCount()` method provides the number of rows of data exposed by the model. We return the length of the `self.lang` list.

```
def flags(self, index):  
  
    fg1 = Qt.ItemIsEnabled  
    fg2 = Qt.ItemIsSelectable  
  
    return fg1 | fg2
```

By returning the `Qt.ItemIsEnabled` and `Qt.ItemIsSelectable` flags, we enable the data item and make it selectable.

```
def initData(self):  
  
    languages = ("Python", "Ruby", "Java",  
                "C", "C#", "C++")  
    self.model = MyListModel(languages)
```

Inside the `initData()` method, we create the instance of our custom data model.

```
lv = QListView(self)  
lv.setModel(self.model)
```

Inside the `initUI()` method, the model is attached to the `QListView` widget.

In the second program we will further enhance our the example. The model is made editable. We can now modify the data in the list view widget.

---

#### Listing 6.9: Subclassing `QAbstractListModel` II

---

```
#!/usr/bin/python3  
# -*- coding: utf-8 -*-  
  
'''  
ZetCode Advanced PyQt5 tutorial  
  
In this example, we create an editable  
QAbstractListModel subclass.  
  
Author: Jan Bodnar  
Website: zetcode.com  
Last edited: August 2017  
'''  
  
from PyQt5.QtWidgets import (QWidget, QApplication,  
                             QListView, QVBoxLayout)  
from PyQt5.QtCore import (Qt, QAbstractListModel,  
                           QModelIndex, QVariant)  
import sys  
  
class MyListModel(QAbstractListModel):
```

```

def __init__(self, lang):
    super().__init__()

    self.lang = lang

def data(self, index, role):

    if index.isValid() and role == Qt.DisplayRole:
        return QVariant(self.lang[index.row()])

    else:
        return QVariant()

def setData(self, index, val, role):

    if not val or not index.isValid():
        return False

    if role == Qt.EditRole:

        self.lang[index.row()] = val
        self.dataChanged.emit(index, index)

        return True

    else: return False

def rowCount(self, index=QModelIndex()):

    return len(self.lang)

def flags(self, index):

    fg1 = Qt.ItemIsEnabled
    fg2 = Qt.ItemIsSelectable
    fg3 = Qt.ItemIsEditable

    return fg1 | fg2 | fg3

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 250, 200)
        self.setWindowTitle("Subclass")

        self.initUI()

    def initUI(self):

        languages = ["Python", "Ruby", "Java",
                     "C", "C#", "C++"]

        self.model = MyListModel(languages)

```

```

        lv = QListView(self)
        lv.setModel(self.model)

        layout = QVBoxLayout()
        layout.addWidget(lv)
        self.setLayout(layout)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

We make our model editable by adding the `Qt.ItemIsEditable` flag and providing a `setData()` method.

```

if not val or not index.isValid():
    return False

```

The `setData()` method receives the item value that we are about to change. We make sure that it is valid and non empty.

```

if role == Qt.EditRole:

    self.lang[index.row()] = val
    self.dataChanged.emit(index, index)

    return True

else: return False

```

For the editable role, we update the data and send a `dataChanged` signal. This informs a view about the modified data.

```

def flags(self, index):

    fg1 = Qt.ItemIsEnabled
    fg2 = Qt.ItemIsSelectable
    fg3 = Qt.ItemIsEditable

    return fg1 | fg2 | fg3

```

We add the `Qt.ItemIsEditable` flag to the flags.

## 6.6 Filtering data with `QSortFilterProxyModel`

`QSortFilterProxyModel` can be used for sorting and filtering items. In the following example, we use this class to filter data.

Listing 6.10: Filtering Data

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we filter items in
a QListView using a QSortFilterProxyModel class.

```

*Author: Jan Bodnar*  
*Website: zetcode.com*  
*Last edited: August 2017*  
,,

```
from PyQt5.QtWidgets import (QWidget, QApplication, QListView,
                              QVBoxLayout, QLineEdit, QCheckBox, QComboBox, QGridLayout)
from PyQt5.QtCore import (Qt, QStringListModel, QRegExp,
                           QVariant, QSortFilterProxyModel)
import sys
```

```
class Example(QWidget):
```

```
    def __init__(self):
        super().__init__()
```

```
        self.setGeometry(300, 300, 400, 240)
        self.setWindowTitle("Filtering data")
```

```
        self.initData()
        self.initUI()
```

```
    def initData(self):
```

```
        words = ["radar", "robert", "Rome", "rodeo",
                  "rust", "ready", "robot", "rampart", "RAM", "ROM"]
```

```
        self.model = QStringListModel(words)
        self.filterModel = QSortFilterProxyModel(self)
        self.filterModel.setSourceModel(self.model)
        self.filterModel.setDynamicSortFilter(True)
```

```
    def initUI(self):
```

```
        grid = QGridLayout()
        grid.setSpacing(10)
```

```
        self.lv = QListView(self)
        self.lv.setModel(self.filterModel)
        grid.addWidget(self.lv, 0, 0, 2, 2)
```

```
        self.filText = QLineEdit(self)
        grid.addWidget(self.filText, 0, 3, Qt.AlignTop)
```

```
        self.case = QCheckBox("Case sensitive", self)
        grid.addWidget(self.case, 1, 3, Qt.AlignTop)
```

```
        self.filterCombo = QComboBox(self)
        self.filterCombo.addItem("Regular expression",
                                  QVariant(QRegExp.RegExp))
        self.filterCombo.addItem("Wildcard",
                                  QVariant(QRegExp.Wildcard))
        self.filterCombo.addItem("Fixed string",
                                  QVariant(QRegExp.FixedString))
        grid.addWidget(self.filterCombo, 2, 0)
```

```
        self.filterCombo.activated[str].connect(self.filterItems)
        self.filText.textChanged[str].connect(self.filterItems)
```



```

        self.case.toggled[bool].connect(self.filterItems)

        self.setLayout(grid)

    def filterItems(self, value):

        idx = self.filterCombo.currentIndex()

        syntaxType = self.filterCombo.itemData(idx)
        syntax = QRegExp.PatternSyntax(syntaxType)

        if self.case.isChecked():
            case = Qt.CaseSensitive

        else:
            case = Qt.CaseInsensitive

        regExp = QRegExp(self.filText.text(), case, syntax)
        self.filterModel.setFilterRegExp(regExp)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

We have ten words in a `QListView` widget. We use a filter to hide items from a list view. We also enable an optional case sensitive filtering.

```

words = ["radar", "robert", "Rome", "rodeo",
         "rust", "ready", "robot", "rampart", "RAM", "ROM"]

```

These lines build the list of strings that appears in the `QListView` widget.

```

self.model = QStringListModel(words)
self.filterModel = QSortFilterProxyModel(self)
self.filterModel.setSourceModel(self.model)

```

We create the `QSortFilterProxyModel` object.

```

self.filterModel.setDynamicSortFilter(True)

```

With the `setDynamicSortFilter()` method we ensure that the model is dynamically sorted and filtered whenever the contents of the source model change.

```

self.filterCombo = QComboBox(self)
self.filterCombo.addItem("Regular expression",
    QVariant(QRegExp.RegExp))
self.filterCombo.addItem("Wildcard",
    QVariant(QRegExp.Wildcard))
self.filterCombo.addItem("Fixed string",
    QVariant(QRegExp.FixedString))
grid.addWidget(self.filterCombo, 2, 0)

```

We create a combo box widget. It has three pattern syntax options: a regular expression, a wildcard, and a fixed string. The regular expression pattern is a Perl-like rich pattern matching syntax. The wildcard pattern matching syntax is well known to users of terminals. The third fixed string is a very simple string matching pattern.

```

self.filterCombo.activated[str].connect(self.filterItems)
self.filText.textChanged[str].connect(self.filterItems)
self.case.toggled[bool].connect(self.filterItems)

```

We call the `filterItems()` method in three cases. When we select an option from a combo box, key in some text in the line edit widget, or click on the check box.

```

idx = self.filterCombo.currentIndex()

syntaxType = self.filterCombo.itemData(idx)
syntax = QRegExp.PatternSyntax(syntaxType)

```

Inside the `filterItems()` method, we determine the pattern syntax type.

```

if self.case.isChecked():
    case = Qt.CaseSensitive

else:
    case = Qt.CaseInsensitive

```

We determine whether the filtering is case sensitive.

```

regExp = QRegExp(self.filText.text(), case, syntax)
self.filterModel.setFilterRegExp(regExp)

```

We create a `QRegExp` object. It is used for pattern matching. We provide a text, case sensitiveness and pattern matching syntax type. Finally, we do the filtering by calling the `setFilterRegExp()` method.

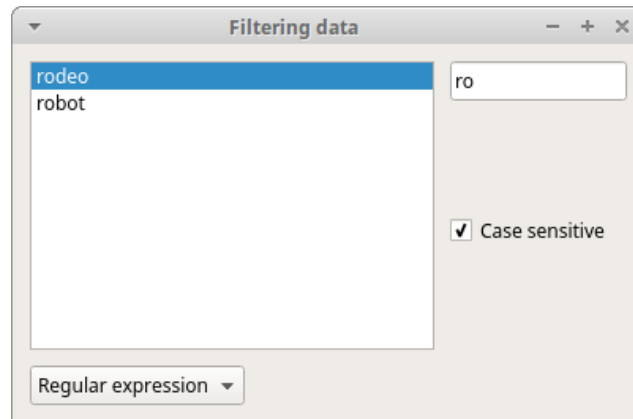


Figure 6.8: Filtering data

Figure 6.8 shows data filtered using case sensitive regular expression pattern matching syntax.

## 6.7 Delegates

The model stores the data. The view presents the data to the user. There is another component called a **delegate**. The delegate provides display and editing facilities for data items from a model. In other words, we can customize the

way the data is displayed and edited. The delegate works as an intermediary between the model and the view.

`QStyledItemDelegate` works as the base class for our custom delegate. When editing data, the delegate provides an editor widget. This widget will pop up when the data item is being modified.

There are four important methods that we need to implement:

- `createEditor()` returns the widget to modify the data
- `setEditorData()` provides the widget with data to manipulate
- `updateEditorGeometry()` ensures that the editor is displayed correctly
- `setModelData()` returns updated data to the model

---

Listing 6.11: A Custom Delegate

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we use a
new delegate in a QListView.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QListView,
                             QVBoxLayout, QSpinBox, QStyledItemDelegate)
from PyQt5.QtCore import Qt, QVariant, QStringListModel
import sys

class MyDelegate(QStyledItemDelegate):

    def __init__(self):
        super().__init__()

    def createEditor(self, parent, option, index):

        editor = QSpinBox(parent)
        editor.setMinimum(0)
        editor.setMaximum(100)

        return editor

    def setEditorData(self, editor, index):

        model = index.model()
        role = Qt.DisplayRole
        value = model.data(index, role)
        editor.setValue(int(value))
```

```

def setModelData(self, editor, model, index):

    value = editor.value()
    model.setData(index, QVariant(value))

def updateEditorGeometry(self, editor, option, index):

    r = option.rect
    r.setHeight(30)
    editor.setGeometry(r)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 250, 200)
        self.setWindowTitle("Delegate")

        self.initData()
        self.initUI()

    def initData(self):

        vals = ["0", "1", "2", "3", "4"]

        self.model = QStringListModel(vals)

    def initUI(self):

        lv = QListView(self)
        lv.setModel(self.model)

        self.de = MyDelegate()
        lv.setItemDelegate(self.de)

        layout = QVBoxLayout()
        layout.addWidget(lv)
        self.setLayout(layout)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

The example shows a list of numbers in a list view widget. We create our custom delegate. When we modify a number, a spin box widget pops up.

```

class MyDelegate(QStyledItemDelegate):

    def __init__(self):
        super().__init__()

```

The custom delegate inherits from the `QStyledItemDelegate` class.

```

def createEditor(self, parent, option, index):

```

```

        editor = QSpinBox(parent)
        editor.setMinimum(0)
        editor.setMaximum(100)

    return editor

```

The `createEditor()` method creates the editor for the delegate. In our case, we create an instance of the `QSpinBox` widget and set its minimum and maximum value.

```

def setEditorData(self, editor, index):

    model = index.model()
    role = Qt.DisplayRole
    value = model.data(index, role)
    editor.setValue(int(value))

```

The `setEditorData()` gives data from the model to the editor.

```

def setModelData(self, editor, model, index):

    value = editor.value()
    model.setData(index, QVariant(value))

```

When we modify the data with the editor, we update the model accordingly.

```

def updateEditorGeometry(self, editor, option, index):

    r = option.rect
    r.setHeight(30)
    editor.setGeometry(r)

```

The `updateEditorGeometry()` method ensures that the editor is displayed correctly. The default size of the editor for our theme is too small. Here we increase the height of the editor.

```

self.de = MyDelegate()
lv.setItemDelegate(self.de)

```

We create an instance of the custom delegate and set the delegate to the list view widget.

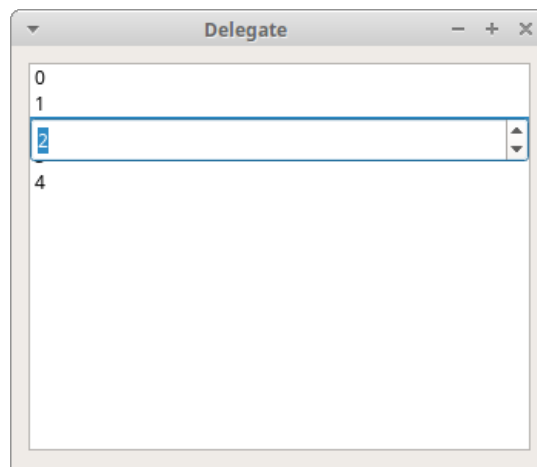


Figure 6.9: A custom delegate

Figure 6.9 shows a spin box widget while editing a number.

In the second example, we change the appearance of items. We change the background color of the selected item and the foreground color of all items.

Listing 6.12: A Custom Delegate II

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we change the background
color of the selected item and the
foreground color of all items using
the QStyledItemDelegate class.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5.QtWidgets import (QWidget, QApplication, QListView,
                             QVBoxLayout, QStyledItemDelegate, QStyle)
from PyQt5.QtGui import QBrush, QColor, QPen
from PyQt5.QtCore import Qt, QStringListModel
import sys

class MyDelegate(QStyledItemDelegate):

    def __init__(self):
        super().__init__()

    def paint(self, painter, option, index):

        painter.setPen(QPen(Qt.NoPen))
```

```

        if option.state & QStyle.State_Selected:
            brush = QBrush(QColor("#66ff71"))
            painter.setBrush(brush)

        else:
            brush = QBrush(Qt.white)
            painter.setBrush(brush)

        painter.drawRect(option.rect)

        text = index.data(Qt.DisplayRole)

        if text:

            painter.setPen(QPen(Qt.blue))
            align = Qt.AlignCenter
            painter.drawText(option.rect, align, text)

class Example(QWidget):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 250, 200)
        self.setWindowTitle("Delegate")

        self.initData()
        self.initUI()

    def initData(self):

        names = ["Jack", "Tom", "Lucy", "Bill", "Jane"]
        self.model = QStringListModel(names)

    def initUI(self):

        lv = QListView(self)
        lv.setModel(self.model)

        self.de = MyDelegate()
        lv.setItemDelegate(self.de)

        layout = QVBoxLayout()
        layout.addWidget(lv)
        self.setLayout(layout)

app = QApplication([])
ex = Example()
ex.show()
sys.exit(app.exec_())

```

---

When we want to customize the drawing of our items, we reimplement the `paint()` method of the delegate. The method is called individually for each item. If we needed to influence the size of the item, we would also reimplement the `sizeHint()` method.

```

def paint(self, painter, option, index):

```

...

The `paint()` method renders the delegate using the given painter and style option for the item specified by index.

```
painter.setPen(QPen(Qt.NoPen))

if option.state & QStyle.State_Selected:
    brush = QBrush(QColor("#66ff71"))
    painter.setBrush(brush)
else:
    brush = QBrush(Qt.white)
    painter.setBrush(brush)

painter.drawRect(option.rect)
```

If the item is selected, we set a green brush to the painter; otherwise we use a plain white color. The `drawRect()` method then draws the background of the delegate with the chosen color.

```
text = index.data(Qt.DisplayRole)
```

We get the actual data of the item from the model.

```
if text:
    painter.setPen(QPen(Qt.blue))
    align = Qt.AlignCenter
    painter.drawText(option.rect, align, text)
```

We draw the text in blue color and align it to the center.

```
self.de = MyDelegate()
lv.setItemDelegate(self.de)
```

We create the instance of our delegate and set it to the list view.

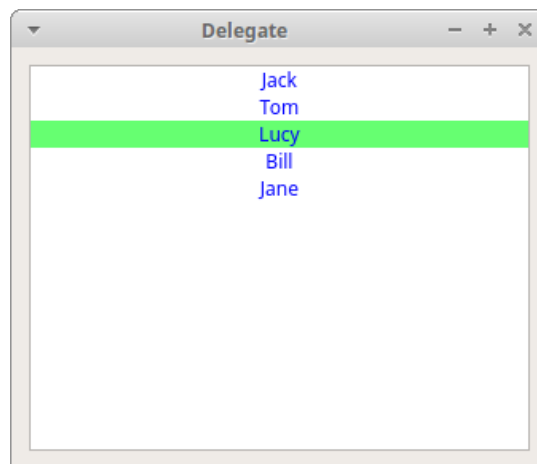


Figure 6.10: Drawing a custom delegate

Figure 6.10 shows data items with customized foreground and background col-



ors.

## Chapter 7

# QtSql module

The QSql module contains classes for working with databases. It supports multiple databases, for example SQLite, Oracle, MySQL, InterBase, DB2, Sybase or PostgreSQL. The QSql module supports them through various database drivers. The database classes are divided into three layers. The driver layer contains classes that provide the low-level bridge between the specific database and the SQL API. The SQL API layer contains classes that interact with the database. They create a database connection, perform queries, handle SQL errors etc. The third layer, the UI layer, links data from the database to the data-aware widgets. This QSql module greatly simplifies the tedious work with the databases. In our examples, we work with the SQLite database. In Python version 2.5 and above, the SQLite database is shipped with the language.

Using QSql as a way of dealing with databases has one disadvantage, too. It does not handle database errors via exceptions. The reason lies in the history of the Qt toolkit.<sup>1</sup> As a result, we need to call the `lastError()` method for each execution of an SQL query.

### 7.1 SQLite database

SQLite is an embedded relational database engine. Its developers call it a self-contained, serverless, zero-configuration and transactional SQL database engine. It is very popular and there are hundreds of millions copies worldwide in use today. SQLite is used in Solaris 10 and Mac OS operating systems, iPhone or Skype. Qt5 library has a built-in support for the SQLite as well as the Python or the PHP language. Many popular applications use SQLite internally such as Firefox or Amarok.

SQLite implements most of the SQL-92 standard for SQL. The SQLite engine is not a standalone process. Instead, it is statically or dynamically linked into the application. SQLite library has a small size. It could take less than 300 KiB. An SQLite database is a single ordinary disk file that can be located anywhere in the directory hierarchy. It is a cross platform file. Can be used on various operating systems, both 32 and 64 bit architectures. SQLite is created in C programming language and has bindings for many languages like C++, Java,

---

<sup>1</sup><http://qt-project.org/forums/viewthread/1150>

C#, Python, Perl, Ruby, Visual Basic, Tcl and others. The source code of SQLite is in public domain.

## 7.2 QSqlQuery

The `QSqlQuery` is a class which is used for executing SQL statements. The class has methods for executing statements, retrieving results, getting error messages, navigating through records, and getting additional information about queries.

### 7.2.1 Database version

In the first example, we determine the version of the SQLite database.

Listing 7.1: Database version

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example prints the version of
the SQLite database.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QSql
import sys

def check_error(q, db):

    ler = q.lastError()

    if ler.isValid():

        print(ler.text())
        db.close()

        sys.exit(1)

def main():

    db = QSql.QSqlDatabase.addDatabase("QSQLITE")
    db.setDatabaseName(":memory:")

    if not db.open():
        print("Cannot establish a database connection")
        sys.exit(1)

    q = QSql.QSqlQuery()
    q.exec_("SELECT sqlite_version()")
    check_error(q, db)

    q.first()
```

```

        print("Database version: {}".format(q.value(0)))

    db.close()

if __name__ == '__main__':
    main()

```

---

In the example, we determine the version of the SQLite database.

```
from PyQt5 import QtSql
```

We import the `QtSql` module; it has classes for working with database systems.

```

def check_error(q, db):

    ler = q.lastError()

    if ler.isValid():

        print(ler.text())
        db.close()

        sys.exit(1)

```

We check if an error occurred in the query. The error information about the last error that turned up with an SQL query is returned by the `lastError()` method. In case of an error, we print the error message, close the database connection, and exit the program.

```
db = QtSql.QSqlDatabase.addDatabase("QSQLITE")
```

We create a database connection. We use a driver for the SQLite database. Note that we merely create a connection object; it is not yet opened.

```
db.setDatabaseName(":memory:")
```

We work with an in-memory database.

```

if not db.open():
    print("Cannot establish a database connection")
    sys.exit(1)

```

The `open()` method opens a database connection. If the connection cannot be opened, it returns `False`.

```
q = QtSql.QSqlQuery()
```

We create an instance of the `QSqlQuery`, which is a class for executing SQL statements.

```
q.exec_("SELECT sqlite_version()")
```

The `exec_()` method executes the query.

```
check_error(q, db)
```

We check for possible errors.

```
q.first()
```

The `first()` method retrieves the first record from the result, if available, and positions the query on the retrieved record. In our case, we only have one record. (The term record is a synonym for a database row.)

```
print("Database version: {}".format(q.value(0)))
```

This line prints the database version to the console. The `value()` method gets the field values from the record. One record consists from one or more fields. The fields are accessed by their index numbers. In our case we have only one field in the record.

```
db.close()
```

In the end, the connection is closed.

```
$ ./version.py
Database version: 3.16.2
```

The `version.py` script prints the version of the SQLite database.

## 7.2.2 Creating a table

The `QtSql` module does not work with database exceptions. For checking errors, we have to call the `lastError()` method of the `QSqlQuery`.

Listing 7.2: Creating a table

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example creates a Cars table and
provides error checking.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QSql
import sys

def check_error(q, db):

    ler = q.lastError()

    if ler.isValid():

        print(ler.text())
        db.close()

        exit(1)

def main():

    db = QSql.QSqlDatabase.addDatabase("QSQLITE")

    db.setDatabaseName("test.db")
```

```

if not db.open():
    print("Cannot establish a database connection")
    sys.exit(1)

q = QSqlQuery()
q.exec_("DROP TABLE IF EXISTS Cars")
check_error(q, db)
q.exec_("CREATE TABLE Cars(Id INT, Name TEXT, Price INT)")
check_error(q, db)
q.exec_("INSERT INTO Cars VALUES(1,'Audi',52642)")
check_error(q, db)
q.exec_("INSERT INTO Cars VALUES(2,'Mercedes',57127)")
check_error(q, db)
q.exec_("INSERT INTO Cars VALUES(3,'Skoda',9000)")
check_error(q, db)
q.exec_("INSERT INTO Cars VALUES(4,'Volvo',29000)")
check_error(q, db)
q.exec_("INSERT INTO Cars VALUES(5,'Bentley',350000)")
check_error(q, db)
q.exec_("INSERT INTO Cars VALUES(6,'Citroen',21000)")
check_error(q, db)
q.exec_("INSERT INTO Cars VALUES(7,'Hummer',41400)")
check_error(q, db)
q.exec_("INSERT INTO Cars VALUES(8,'Volkswagen',21600)")
check_error(q, db)

db.close()

if __name__ == '__main__':
    main()

```

---

The script creates a `Cars` table. It provides error handling.

```

q.exec_("DROP TABLE IF EXISTS Cars")
check_error(q, db)

```

Each execution of an SQL query is followed by the `check_error()` function call.

### 7.2.3 Selecting rows

In the following example, we retrieve multiple rows from the database table. We navigate through records with the `next()` method.

---

Listing 7.3: Selecting rows

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example selects five rows from
a Cars table.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

```

```

from PyQt5 import QSql
import sys

def check_error(q, db):

    ler = q.lastError()

    if ler.isValid():

        db.close()
        print(ler.text())
        sys.exit(1)

def main():

    db = QSql.QSqlDatabase.addDatabase("QSQLITE")
    db.setDatabaseName("test.db")

    if not db.open():
        print("Cannot establish a database connection")
        sys.exit(1)

    q = QSql.QSqlQuery()
    q.exec_("SELECT * FROM Cars LIMIT 5")
    check_error(q, db)

    while q.next():

        cid = q.value(0)
        name = q.value(1)
        price = q.value(2)

        print(cid, name, price)

    db.close()

if __name__ == '__main__':

    main()

```

---

In the example we select five cars from the Cars table.

```
q.exec_("SELECT * FROM Cars LIMIT 5")
```

This is the query for retrieving five rows.

```

while q.next():

    cid = q.value(0)
    name = q.value(1)
    price = q.value(2)

    print(cid, name, price)

```

The `next()` method retrieves the next record from the result. If there are no more records to retrieve, a Boolean false is returned. We get three fields from each record. Each field has its index in the record. The fields are picked by the `value()` method of the `QSqlQuery` class.

```
$ ./select_cars.py
1 Audi 52642
2 Mercedes 57127
3 Skoda 9000
4 Volvo 29000
5 Bentley 350000
```

The output shows five cars from the `Cars` table.

## 7.2.4 Prepared statements

Parametrized or prepared statements increase security and performance of SQL queries. They guard against infamous SQL injections. We use the `prepare()` and `bindValue()` methods of the `QSqlQuery` class to create prepared statements.

Listing 7.4: Prepared statements

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example we work with a prepared
SQL statement.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QSql
import sys

def check_error(q, db):

    ler = q.lastError()

    if ler.isValid():

        db.close()
        print(ler.text())
        sys.exit(1)

def main():

    db = QSql.QSqlDatabase.addDatabase("QSQLITE")
    db.setDatabaseName("test.db")

    if not db.open():
        print("Cannot establish a database connection")
        sys.exit(1)

    q = QSql.QSqlQuery()
    q.prepare("SELECT Name, Price FROM Cars WHERE Id = ?")
    q.bindValue(0, 6)
    q.exec_()
    check_error(q, db)
```



```

        q.first()

        name = q.value(0)
        price = q.value(1)
        print(name, price)

    db.close()

if __name__ == '__main__':
    main()

```

---

We select a specific car from the Cars table using a prepared statement.

```
q.prepare("SELECT Name, Price FROM Cars WHERE Id = ?")
```

The `prepare()` method prepares the SQL query for execution. We use a question mark placeholder, which is later bound to a specific value.

```
q.bindValue(0, 6)
```

A value is bound to the placeholder using the `bindValue()` method.

```
q.exec_()
```

The query is executed.

## 7.3 Metadata

Metadata is data about the data in the database tables. Examples of metadata include table names, column names, or a database version. We have three examples relating to metadata.

### 7.3.1 Driver features

The driver is performing a conversation with a specific database system. We can ask the driver for supported features.

---

Listing 7.5: Driver features

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example shows available SQLite driver
features.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QtSql
import sys

```

```
def main():

    db = QSql.QSqlDatabase.addDatabase("QSQLITE")

    driver = db.driver()
    print(driver.hasFeature(QtSql.QSqlDriver.Transactions))
    print(driver.hasFeature(QtSql.QSqlDriver.BLOB))
    print(driver.hasFeature(QtSql.QSqlDriver.Unicode))
    print("*****")
    print(driver.hasFeature(QtSql.QSqlDriver.PreparedQueries))
    print(driver.hasFeature(QtSql.QSqlDriver.NamedPlaceholders))
    print(driver.hasFeature(QtSql.QSqlDriver.MultipleResultSets))
    print("*****")
    print(driver.hasFeature(QtSql.QSqlDriver.QuerySize))
    print(driver.hasFeature(QtSql.QSqlDriver.BatchOperations))
    print(driver.hasFeature(QtSql.QSqlDriver.EventNotifications))

    db.close()

if __name__ == '__main__':

    main()
```

---

We request the driver for twelve features for the SQLite database.

```
db = QSql.QSqlDatabase.addDatabase("QSQLITE")
```

A driver for the SQLite database is used.

```
driver = db.driver()
```

We get the driver object.

```
print(driver.hasFeature(QtSql.QSqlDriver.Transactions))
```

The `hasFeature()` method is used to figure out whether the database supports transactions.

```
$ ./driver_features.py
True
True
True
*****
True
False
False
*****
False
False
True
```

From the output of the example we can see that the SQLite database supports transactions, BLOB data type, Unicode strings, prepared queries, and event notifications. It does not support named placeholders, multiple result sets, reporting the size of a query, batch operations, and event notifications.

### 7.3.2 Column names

The second example deals with column names of the returned data.

Listing 7.6: Column names

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example shows column names of data.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QSql
import sys

def check_error(q, db):

    ler = q.lastError()

    if ler.isValid():
        print(ler.text() )

def main():

    db = QSql.QSqlDatabase.addDatabase("QSQLITE")
    db.setDatabaseName("test.db")

    if not db.open():
        print("Cannot establish a database connection")
        sys.exit(1)

    q = QSql.QSqlQuery()
    q.exec_("SELECT * FROM Cars LIMIT 1")
    check_error(q, db)

    rec = q.record()

    print("Number of columns:", rec.count())

    print(rec.fieldName(0))
    print(rec.fieldName(1))
    print(rec.fieldName(2))

    db.close()

if __name__ == '__main__':

    main()
```

---

The code example prints the columns of the **Cars** table.

```
q = QSql.QSqlQuery()
q.exec_("SELECT * FROM Cars LIMIT 1")
```

The **SELECT** statement takes all three columns for one row of the **Cars** table.

```
rec = q.record()
```

The `record()` method returns the `QSqlRecord` object which encapsulates the metadata information about a query.

```
print("Number of columns:", rec.count())
```

The number of columns is determined with the `count()` method.

```
print("Column names:")
print(rec.fieldName(0))
print(rec.fieldName(1))
print(rec.fieldName(2))
```

Using the `fieldName()` method of the record object, we get the names of the columns of the query.

```
$ ./column_names.py
Number of columns: 3
Column names:
Id
Name
Price
```

The script prints the number of columns and their names to the console.

### 7.3.3 Affected rows

The number of rows affected by a query belongs to metadata as well. To get the number of affected rows, we use the `numRowsAffected()` method of the `QSqlQuery` class.

---

Listing 7.7: Number of rows affected

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example we determine the number of rows
affected by a query.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QSql
import sys

def check_error(q, db):

    ler = q.lastError()

    if ler.isValid():

        db.close()
        print(ler.text())
```

```

        sys.exit(1)

def main():

    db = QSql.QSqlDatabase.addDatabase("QSQLITE")
    db.setDatabaseName("test.db")

    if not db.open():
        print("Cannot establish a database connection")
        sys.exit(1)

    q = QSql.QSqlQuery()
    q.exec_("DELETE FROM Cars WHERE Id IN (1, 2, 3)")
    check_error(q, db)

    print("The query affected {0} rows".format(
        q.numRowsAffected()))
    db.close()

if __name__ == '__main__':

    main()

```

---

In the script we determine the number of rows that were removed by the DELETE statement.

```

q = QSql.QSqlQuery()
q.exec_("DELETE FROM Cars WHERE Id IN (1, 2, 3)")

```

The DELETE statement removes the first three rows of the `Cars` table.

```

print("The query affected {0} rows".format(q.numRowsAffected()))

```

We get the number of rows affected by our query using the `numRowsAffected()` method.

```

$ ./rows_affected.py
The query affected 3 rows
$ ./rows_affected.py
The query affected 0 rows

```

In the first execution, the SQL query has influenced three rows. Subsequent execution of the same script has not changed any additional rows; they were already deleted.

## 7.4 Transactions

If we want to have all or nothing outcome, we can use transactions. A transaction is an atomic unit of database operations against the data in one or more databases. The effects of all the SQL statements in a transaction can be either all committed to the database or all rolled back.

To work with a transaction in the `QSql` module, we use the `transaction()`, `commit()` and `rollback()` methods of the `QSqlDatabase` class.

---

Listing 7.8: Transaction

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example works with a transaction.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QtSql
import sys

def check_error(q, db):

    ler = q.lastError()

    if ler.isValid():

        print(ler.text())

        db.rollback()
        db.close()

        sys.exit(1)

def main():

    db = QtSql.QSqlDatabase.addDatabase("QSQLITE")

    db.setDatabaseName("test.db")

    if not db.open():
        print("cannot establish a database connection")
        sys.exit(1)

    db.transaction()

    q = QtSql.QSqlQuery()

    q.exec_("DROP TABLE IF EXISTS Cars")
    check_error(q, db)

    q.exec_("CREATE TABLE Cars(Id INT,"
        "Name TEXT, Price INT)")
    check_error(q, db)

    q.exec_("INSERT INTO Cars VALUES(1,'Audi ',52642)")
    check_error(q, db)

    q.exec_("INSERT INTO Cars VALUES(2,'Mercedes ',57127)")
    check_error(q, db)

    q.exec_("INSERT INTO Cars VALUES(3,'Skoda ',9000)")
    check_error(q, db)

    q.exec_("INSERT INTO Cars VALUES(4,'Volvo ',29000)")
    check_error(q, db)

```

```

q.exec_("INSERT INTO Cars VALUES(5,'Bentley',350000)")
check_error(q, db)

q.exec_("INSERT INTO Cars VALUES(6,'Citroen',21000)")
check_error(q, db)

q.exec_("INSERT INTO Cars VALUES(7,'Hummer',41400)")
check_error(q, db)

q.exec_("INSERT INTO Cars VALUES(8,'Volkswagen',21600)")
check_error(q, db)

db.commit()

db.close()

if __name__ == '__main__':
    main()

```

---

The example demonstrates how to create the `Cars` table in a single transaction.

```

def check_error(q, db):
    ler = q.lastError()

    if ler.isValid():
        print(ler.text())

        db.rollback()
        db.close()

        sys.exit(1)

```

In case of an error, we call the `rollback()` method and nothing is saved.

```
db.transaction()
```

A new transaction is started with the `transaction()` method.

```
db.commit()
```

If everything goes OK, we commit the changes to the database with the `commit()` method.

## 7.5 Model and View

In the Model and View chapter we were talking about the Model and View programming paradigm. The `QtSql` module also has some classes that utilize this paradigm.

The `QtSql` module has three models available:

- `QSqlQueryModel`
- `QSqlTableModel`
- `QSqlRelationalTableModel`

The `QSqlQueryModel` provides a read-only data model for SQL result sets. The `QSqlTableModel` provides an editable data model for a single database table. And the `QSqlRelationalTableModel` provides an editable data model for a single database table with foreign key support.

### 7.5.1 Read-only model

The following example creates a small GUI application.

Listing 7.9: Read-only model

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example displays read-only data from a SQLite
database in a QTableView widget.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QtSql
from PyQt5.QtWidgets import QMainWindow, QTableView, \
    QApplication, QAbstractItemView
from PyQt5.QtCore import Qt
import sys

class Example(QMainWindow):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 400, 330)
        self.setWindowTitle("Read-only model")

        self.createConnection()
        self.createModel()
        self.initUI()

        self.statusBar().showMessage("Ready")

    def onClicked(self, index):

        self.statusBar().showMessage(str(index.data()))

    def createConnection(self):

        self.db = QtSql.QSqlDatabase.addDatabase("QSQLITE")
        self.db.setDatabaseName("test.db")

        if not self.db.open():
            print("Cannot establish a database connection")
            return False
```



```

def createModel(self):

    self.model = QSql.QSqlQueryModel()
    query = QSql.QSqlQuery()
    query.exec_("SELECT * FROM Cars")

    self.model.setQuery(query)
    self.model.removeColumn(0)

    self.model.setHeaderData(0, Qt.Horizontal, "Name")
    self.model.setHeaderData(1, Qt.Horizontal, "Price")

def initUI(self):

    self.view = QTableView()
    self.view.setModel(self.model)

    mode = QAbstractItemView.SingleSelection
    self.view.setSelectionMode(mode)

    self.view.clicked.connect(self.onClicked)
    self.setCentralWidget(self.view)

def closeEvent(self, e):

    if self.db.open():
        self.db.close()

def main():

    app = QApplication([])
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

---

We show the data from the `Cars` table in the `QTableView` widget. The widget serves as the view and the `QSqlQueryModel` as the model. This model is a read-only model, so the data cannot be modified.

```

class Example(QMainWindow):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 400, 330)
        self.setWindowTitle("Read-only model")
        ...

```

The main application window inherits from the `QMainWindow`.

```

self.createConnection()
self.createModel()
self.initUI()

```

Three methods perform three different tasks. First, we create a connection to

the database. Then we create a data model. Finally, we present the data from the model in a view widget.

```
def onClicked(self, index):  
  
    self.statusBar().showMessage(str(index.data()))
```

Each time we click on the table, the `onClicked()` method is called. The method shows the contents of the currently selected cell in the statusbar.

```
def createConnection(self):  
  
    self.db = QSqlDatabase.addDatabase("QSQLITE")  
    self.db.setDatabaseName("test.db")  
  
    if not self.db.open():  
        print("Cannot establish a database connection")  
        return False
```

These lines establish a connection to the `test.db` database.

```
def createModel(self):  
  
    self.model = QSqlQueryModel()  
    query = QSqlQuery()  
    query.exec_("SELECT * FROM Cars")  
  
    self.model.setQuery(query)  
    self.model.removeColumn(0)  
    ...
```

We create an instance of the read-only `QSqlQueryModel` in the `createModel()` method. We create a query that retrieves all data from the `Cars` table. We set this query to the model.

```
self.model.removeColumn(0)
```

We remove the first column from the model. We do not want to display the ids of the table records. Another way to get the same result would be to select only `Name` and `Price` columns of the table.

```
self.model.setHeaderData(0, Qt.Horizontal, "Name")  
self.model.setHeaderData(1, Qt.Horizontal, "Price")
```

Here we set table headers for the table columns of the `QTableView`.

```
def initUI(self):  
  
    self.view = QTableView()  
    self.view.setModel(self.model)  
    ...
```

In the method we create an instance of the `QTableView` widget and set the model to it. Now there is a bridge between the view and the model.

```
mode = QAbstractItemView.SingleSelection  
self.view.setSelectionMode(mode)
```

These two lines ensure that only one cell of the table widget can be selected at a given time.

```
self.view.clicked.connect(self.onClicked)
```

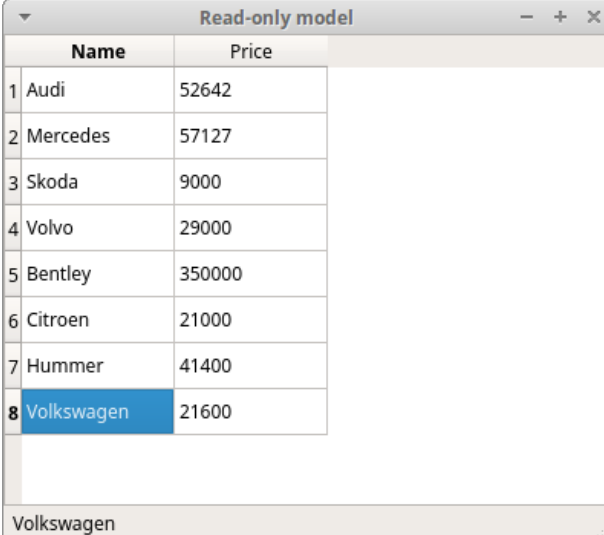
The `onClicked()` method is connected to the clicked signal. This signal is emitted when we click on the `QTableView` widget with a mouse.

```
self.setCentralWidget(self.view)
```

The `QTableView` widget is set to be the central widget of the main application window.

```
def main():  
    app = QApplication([])  
    ex = Example()  
    ex.show()  
    sys.exit(app.exec_())
```

In the `main()` method, we set up and run the application.



|   | Name       | Price  |
|---|------------|--------|
| 1 | Audi       | 52642  |
| 2 | Mercedes   | 57127  |
| 3 | Skoda      | 9000   |
| 4 | Volvo      | 29000  |
| 5 | Bentley    | 350000 |
| 6 | Citroen    | 21000  |
| 7 | Hummer     | 41400  |
| 8 | Volkswagen | 21600  |

Volkswagen

Figure 7.1: Read-only model

### 7.5.2 Editable model

In the second example, we create a model that allows data to be modified. An editable model is created with `QSqlTableModel`. Edit strategies control the way data can be modified in a model. There are three strategies:

- **OnFieldChange** - All changes are applied immediately to the database.
- **OnRowChange** - All changes to a row will be applied when the user selects a different row.
- **OnManualSubmit** - All changes will be cached in the model until either `QSqlTableModel.submitAll()` or `QSqlTableModel.revertAll()` is called.

The default edit strategy is **OnRowChange**.

Listing 7.10: Editable model

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example displays editable data from a SQLite
database in a QTableView widget.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QtSql
from PyQt5.QtWidgets import QMainWindow, QTableView,
    QApplication, QAbstractItemView
from PyQt5.QtCore import Qt
import sys

class Example(QMainWindow):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 400, 330)
        self.setWindowTitle("Editable model")

        self.createConnection()
        self.createModel()
        self.initUI()

        self.statusBar().showMessage("Ready")

    def onClicked(self, index):

        self.statusBar().showMessage(str(index.data()))

    def createConnection(self):

        self.db = QtSql.QSqlDatabase.addDatabase("QSQLITE")
        self.db.setDatabaseName("test.db")

        if not self.db.open():

            print("Cannot establish a database connection")
            return False

    def createModel(self):

        self.model = QtSql.QSqlTableModel()
        self.model.setTable("Cars")
        self.model.select()

        self.model.setHeaderData(0, Qt.Horizontal, "Id")
        self.model.setHeaderData(1, Qt.Horizontal, "Name")
        self.model.setHeaderData(2, Qt.Horizontal, "Price")
```

```

        strategy = QSql.QSqlTableModel.OnFieldChange
        self.model.setEditStrategy(strategy)

    def initUI(self):

        self.view = QTableView()
        self.view.setModel(self.model)

        mode = QAbstractItemView.SingleSelection
        self.view.setSelectionMode(mode)

        self.view.clicked.connect(self.onClicked)

        self.setCentralWidget(self.view)

    def closeEvent(self, e):

        if (self.db.open()):
            self.db.close()

def main():

    app = QApplication([])
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

---

In this example, we show data from the `Cars` table in the `QTableView` widget and use the `QSqlTableModel`. Data can be modified and the changes are applied to the database.

```
self.model = QSql.QSqlTableModel()
```

An instance of the `QSqlTableModel` class is created. This model enables to modify the data.

```
self.model.setTable("Cars")
self.model.select()
```

The model is populated with the data from the `Cars` table.

```
strategy = QSql.QSqlTableModel.OnFieldChange
self.model.setEditStrategy(strategy)
```

Here we set the edit strategy. In this strategy, the changes to the data are immediately applied to the database table.

### 7.5.3 Relational model

In the third example, we use a `QSqlRelationalTableModel`. Is is an editable data model for a single database table. This model has support for foreign keys. By default, changes to a row in the table will be applied when the user selects a different row.

```

-- SQL code for the Authors and Books tables

BEGIN TRANSACTION;
DROP TABLE IF EXISTS Authors;
CREATE TABLE Authors(AuthorId INTEGER PRIMARY KEY, Name TEXT);
INSERT INTO Authors VALUES(1, 'Jane Austen');
INSERT INTO Authors VALUES(2, 'Leo Tolstoy');
INSERT INTO Authors VALUES(3, 'Joseph Heller');
INSERT INTO Authors VALUES(4, 'Charles Dickens');
COMMIT;

BEGIN TRANSACTION;
DROP TABLE IF EXISTS Books;
CREATE TABLE Books(BookId INTEGER PRIMARY KEY, Title TEXT,
    AuthorId INTEGER, FOREIGN KEY(AuthorId)
    REFERENCES Authors(AuthorId));
--REFERENCES Authors(AuthorId) ON DELETE SET NULL);
INSERT INTO Books VALUES(1, 'Emma', 1);
INSERT INTO Books VALUES(2, 'War and Peace', 2);
INSERT INTO Books VALUES(3, 'Catch XXII', 3);
INSERT INTO Books VALUES(4, 'David Copperfield', 4);
INSERT INTO Books VALUES(5, 'Good as Gold', 3);
INSERT INTO Books VALUES(6, 'Anna Karenina', 2);
INSERT INTO Books VALUES(7, 'Pride and Prejudice', 1);
INSERT INTO Books VALUES(8, 'Sense and Sensibility', 1);
COMMIT;

```

To run this example, we need to create two tables: Authors and Books.

Listing 7.11: Relational model

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This example uses the QSqlRelationalTableModel
to display relationship between to tables.

Author: Jan Bodnar
Website: zetcode.com
Last edited: August 2017
'''

from PyQt5 import QSql
from PyQt5.QtWidgets import (QMainWindow, QTableView,
    QApplication, QAbstractItemView)
from PyQt5.QtCore import Qt
import sys

class Example(QMainWindow):

    def __init__(self):
        super().__init__()

        self.setGeometry(300, 300, 400, 330)
        self.setWindowTitle("Relational model")

        self.createConnection()

```

```

        self.createModel()
        self.initUI()

        self.statusBar().showMessage("Ready")

def onClicked(self, index):

    self.statusBar().showMessage(index.data())

def createConnection(self):

    self.db = QSql.QSqlDatabase.addDatabase("QSQLITE")
    self.db.setDatabaseName("test.db")

    if not self.db.open():
        print("Cannot establish a database connection")
        return False

def createModel(self):

    self.model = QSql.QSqlRelationalTableModel()
    self.model.setTable("Books")

    self.model.setHeaderData(0, Qt.Horizontal, "BookId")
    self.model.setHeaderData(1, Qt.Horizontal, "Title")
    self.model.setHeaderData(2, Qt.Horizontal, "Author")

    self.model.setRelation(2, QSql.QSqlRelation("Authors",
        "AuthorId", "Name"))

    self.model.select()

def initUI(self):

    self.view = QTableView()
    self.view.setModel(self.model)

    mode = QAbstractItemView.SingleSelection
    self.view.setSelectionMode(mode)

    self.view.clicked.connect(self.onClicked)

    self.setCentralWidget(self.view)

def closeEvent(self, e):

    if (self.db.open()):
        self.db.close()

def main():

    app = QApplication([])
    ex = Example()
    ex.show()
    sys.exit(app.exec_())

```

```
if __name__ == '__main__':
    main()
```

We use two tables: **Books** and **Authors**. The **AuthorId** column of the **Books** table is a foreign key to the **Id** column of the **Authors** table. If we change some field and select a different row, the changes will be applied to the database.

```
self.model = QSql.QSqlRelationalTableModel()
```

An instance of the `QSqlRelationalTableModel` class is created.

```
self.model.setTable("Books")
```

We set the table on which the model operates.

```
self.model.setHeaderData(0, Qt.Horizontal, "BookId")
self.model.setHeaderData(1, Qt.Horizontal, "Title")
self.model.setHeaderData(2, Qt.Horizontal, "Author")
```

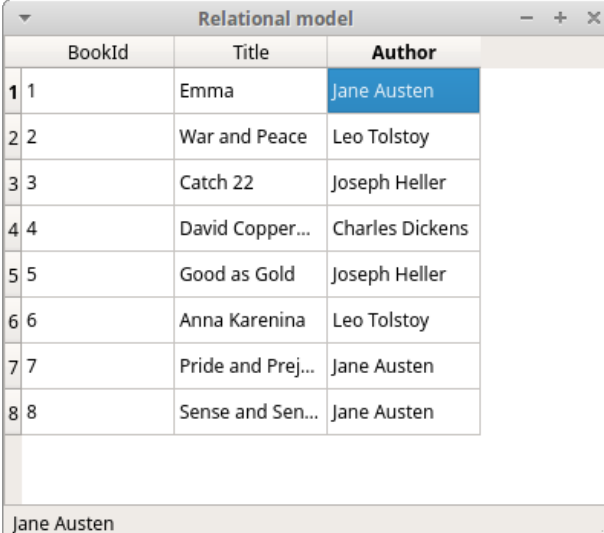
We display three columns in the view widget. The author will be pulled from the **Authors** table by the model.

```
self.model.setRelation(2, QSql.QSqlRelation("Authors",
    "AuthorId", "Name"))
```

Using the `setRelation()` method, we map column 2 of the **Books** table to the **AuthorId** column of the **Authors** table. The last parameter tells the model to display the **Name** column, instead of the **Id** column.

```
self.model.select()
```

The `select()` method populates the model with data from the table set by the `setTable()` method. It is the **Books** table in our case.



|   | BookId | Title             | Author          |
|---|--------|-------------------|-----------------|
| 1 | 1      | Emma              | Jane Austen     |
| 2 | 2      | War and Peace     | Leo Tolstoy     |
| 3 | 3      | Catch 22          | Joseph Heller   |
| 4 | 4      | David Copper...   | Charles Dickens |
| 5 | 5      | Good as Gold      | Joseph Heller   |
| 6 | 6      | Anna Karenina     | Leo Tolstoy     |
| 7 | 7      | Pride and Prej... | Jane Austen     |
| 8 | 8      | Sense and Sen...  | Jane Austen     |

Jane Austen

Figure 7.2: Relational model



## Chapter 8

# Networking

The `QtNetwork` module contains classes that make network programming easier and portable. The module contains lower-level classes such as `QTcpSocket`, `QTcpServer` and `QUdpSocket` that represent low level network concepts. The `QNetworkAccessManager`, on the other hand, is a high-level class that performs network operations using common protocols.

In this chapter, we will do the following:

- Find out all network interfaces.
- Look up an IP address/host name.
- Send HTTP GET, POST, and HEAD requests.
- Authenticate to a web page.
- Fetch an icon from a web page.
- Create a blocking and non-blocking network example.
- Create an Echo client/server example.
- Create a skeleton of a web server.
- Connect to an SMTP server.
- Determine weather conditions using a web service.

### 8.1 Network interfaces

A Network interface is an interface between two pieces of equipment or protocol layers in a network. It is generally a network interface card, but it can be also implemented as a software program. For example, the loopback interface (127.0.0.1 for IPv4 and ::1 for IPv6) is not a physical device but a piece of software simulating a network interface. A network interface will usually have some form of network address.

The `QNetworkInterface` class provides a listing of the host's IP addresses and network interfaces.

## Listing 8.1: Network interfaces

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we find out all
network interfaces on a host.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

import sys
from PyQt5 import QtNetwork

ai = QtNetwork.QNetworkInterface.allInterfaces()

for i in ai:

    print("Interface:", i.name())
    print("Hardware address:", i.hardwareAddress())
    print("IP address(es): ")

    ae = i.addressEntries()

    if len(ae) > 0:

        for e in ae:
            print(e.ip().toString())

        print()
```

---

The example prints all network interfaces on a computer. Modern computers often have three network interfaces: an Ethernet card, a Wifi card, and a local loop.

```
ai = QtNetwork.QNetworkInterface.allInterfaces()
```

The `allInterfaces()` static method returns a listing of all the network interfaces found on the host machine.

```
print("Interface:", i.name())
```

The `name()` method returns the name of the interface. Unix systems have names like `eth0`, `lo`, or `wlan0`. On Windows systems, an interface name is an internal Id.

```
print("Hardware address:", i.hardwareAddress())
```

We determine the hardware address of the interface with `hardwareAddress()`. Each interface has one unique hardware address.

```
ae = i.addressEntries()
```

```
if len(ae) > 0:
```

```
    for e in ae:
```

```

        print(e.ip().toString())

    print()

```

The `addressEntries()` method returns the list of IP addresses that this interface possesses.

## 8.2 IP address, host name

The following two scripts will do simple lookup services. The first one determines the IP address of a domain name. The second one determines a domain name of an IP address. For this, we use the `QHostInfo` class.

---

Listing 8.2: Host name

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we look up a host name
for a given IP address.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtNetwork
from PyQt5.QtCore import QObject, QCoreApplication
import sys

class Example(QObject):

    def __init__(self):
        super().__init__()

        self.doLookup()

    def doLookup(self):

        args = sys.argv

        if len(args) > 1:
            ip = args[1]

        else: ip = '127.0.0.1'

        QtNetwork.QHostInfo.lookupHost(ip, self.receive)

    def receive(self, info):

        er = info.error()

        if er == QtNetwork.QNetworkReply.NoError:

```

```

        print(info.hostName())

    else:
        print("error occurred", er)

    QApplication.quit()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

The script expects an IP address as an argument.

```

args = sys.argv

if len(args) > 1:
    ip = args[1]

else: ip = '127.0.0.1'

```

We retrieve the argument from the command line. If no argument is provided, we use the IP address of a localhost.

```

QtNetwork.QHostInfo.lookupHost(ip, self.receive)

```

The `lookupHost()` looks up the IP address associated with the given host name. When the result of the lookup is ready, the `self.receive` method is called.

```

def receive(self, info):

    er = info.error()

    if er == QtNetwork.QNetworkReply.NoError:

        print(info.hostName())

    else:
        print("error occurred", er)

    QApplication.quit()

```

The `error()` method returns the type of error that occurred if the host name lookup failed; otherwise it returns `QNetworkReply.NoError`. If there is no error, we print the host name with the `hostName()` method. As a final step, we quit the application with the `QCoreApplication.quit()` method.

```

$ ./get_hostname.py 127.0.0.1
localhost

```

This is the result of the `get_hostname.py` program.

The second script looks up the IP address for a given host name.

---

#### Listing 8.3: IP address

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

```

*In this example, we look up the IP address for a given host name.*

*Author: Jan Bodnar  
Website: [zetcode.com](http://zetcode.com)  
Last edited: September 2017  
,,,*

```
from PyQt5 import QtNetwork
from PyQt5.QtCore import QObject, QApplication
import sys

class Example(QObject):

    def __init__(self):
        super().__init__()

        self.doLookup()

    def doLookup(self):

        args = sys.argv

        if len(args) > 1:
            host = args[1]

        else: host = 'localhost'

        QtNetwork.QHostInfo.lookupHost(host, self.receive)

    def receive(self, info):

        er = info.error()

        if er == QtNetwork.QNetworkReply.NoError:

            for ip in info.addresses():
                print(ip.toString())

        else:
            print("error occured", er)

        QApplication.quit()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())
```

---

The script expects a host name as an argument.

```
if er == QtNetwork.QNetworkReply.NoError:

    for ip in info.addresses():
        print(ip.toString())
```

The `addresses()` method returns the list of IP addresses associated with a host name.

```
$ ./get_ipaddress.py wikipedia.org
2620:0:862:ed1a::1
91.198.174.192
```

We get the IP address for the wikipedia.org domain. We get the address in the IPv4 and IPv6 formats.

## 8.3 QNetworkAccessManager

`QNetworkAccessManager` allows the application to send network requests and receive replies. The `QNetworkRequest` holds a request to be sent with the network manager and the `QNetworkReply` contains the data and headers returned for a response.

`QNetworkAccessManager` has an asynchronous API which means that its methods always return immediately and do not wait until they finish. Instead, a signal is emitted when the request is done. We handle the response in the method attached to the `finished` signal.

### 8.3.1 HTTP GET request

The HTTP GET method requests a representation of the specified resource.

Listing 8.4: HTTP GET request

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example we get a web page.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtNetwork
from PyQt5.QtCore import QCoreApplication, QUrl
import sys

class Example:

    def __init__(self):

        self.doRequest()

    def doRequest(self):

        url = "http://www.something.com"
        req = QtNetwork.QNetworkRequest(QUrl(url))

        self.nam = QtNetwork.QNetworkAccessManager()
        self.nam.finished.connect(self.handleResponse)
        self.nam.get(req)
```

```

def handleResponse(self, reply):

    er = reply.error()

    if er == QtNetwork.QNetworkReply.NoError:

        bytes_string = reply.readAll()
        print(str(bytes_string, 'utf-8'))

    else:
        print("Error occurred: ", er)
        print(reply.errorString())

    QApplication.quit()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

The example retrieves the HTML code of the specified web page.

```

url = "http://www.something.com"
req = QtNetwork.QNetworkRequest(QUrl(url))

```

With the `QNetworkRequest` we send a request to the specified URL.

```

self.nam = QtNetwork.QNetworkAccessManager()
self.nam.finished.connect(self.handleResponse)
self.nam.get(req)

```

A `QNetworkAccessManager` object is created. When the request is finished, the `handleResponse()` method is called. The request is fired with the `get()` method.

```

def handleResponse(self, reply):

    er = reply.error()

    if er == QtNetwork.QNetworkReply.NoError:

        bytes_string = reply.readAll()
        print(str(bytes_string, 'utf-8'))

    else:
        print("Error occurred: ", er)
        print(reply.errorString())

    QApplication.quit()

```

The `handleResponse()` receives a `QNetworkReply` object. It contains data and headers for the request that was sent. If there is no error in the network reply, we read all data using the `readAll()` method; otherwise we print an error message. The `errorString()` returns a human-readable description of the last error that occurred. The `readAll()` returns the data in `QByteArray` that has to be decoded.

### 8.3.2 HTTP POST request

The HTTP POST method sends data to the server. The type of the body of the request is indicated by the Content-Type header. A POST request is typically sent via an HTML form. The data sent in the request can be encoded in different ways; in `application/x-www-form-urlencoded` the values are encoded in key-value tuples separated by '&', with a '=' between the key and the value. Non-alphanumeric characters are percent encoded. The `multipart/form-data` is used for binary data and file uploads.

Listing 8.5: HTTP POST request

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example we post data to a web page.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtCore, QtGui, QtNetwork
import sys, json

class Example:

    def __init__(self):

        self.doRequest()

    def doRequest(self):

        data = QtCore.QByteArray()
        data.append("name=Peter&")
        data.append("age=34")

        url = "https://httpbin.org/post"
        req = QtNetwork.QNetworkRequest(QtCore.QUrl(url))
        req.setHeader(QtNetwork.QNetworkRequest.ContentTypeHeader,
            "application/x-www-form-urlencoded")

        self.nam = QtNetwork.QNetworkAccessManager()
        self.nam.finished.connect(self.handleResponse)
        self.nam.post(req, data)

    def handleResponse(self, reply):

        er = reply.error()

        if er == QtNetwork.QNetworkReply.NoError:

            bytes_string = reply.readAll()

            json_ar = json.loads(str(bytes_string, 'utf-8'))
```



```

        data = json_ar['form']

        print('Name: {}'.format(data['name']))
        print('Age: {}'.format(data['age']))

        print()

    else:
        print("Error occurred: ", er)
        print(reply.errorString())

    QtCore.QCoreApplication.quit()

app = QtCore.QCoreApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

The example sends a post request to the <https://httpbin.org/post> testing site, which sends the data back in JSON format.

```

data = QtCore.QByteArray()
data.append("name=Peter&")
data.append("age=34")

```

As per specification, we encode the data sent in the `QByteArray`.

```

req = QtNetwork.QNetworkRequest(QtCore.QUrl(url))
req.setHeader(QtNetwork.QNetworkRequest.ContentTypeHeader,
              "application/x-www-form-urlencoded")

```

We specify the `application/x-www-form-urlencoded` encoding type.

```

bytes_string = reply.readAll()

json_ar = json.loads(str(bytes_string, 'utf-8'))
data = json_ar['form']

print('Name: {}'.format(data['name']))
print('Age: {}'.format(data['age']))

```

In the handler method, we read the response data and decode it. With the built-in `json` module, we extract the posted data.

### 8.3.3 HTTP HEAD request

The HTTP HEAD method allows the client to query the server for the headers for a given resource without actually downloading the resource itself.

---

Listing 8.6: HTTP HEAD request

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example we create a HEAD request.

Author: Jan Bodnar

```

*Website: zetcode.com*  
*Last edited: September 2017*  
,,

```
from PyQt5 import QtNetwork
from PyQt5.QtCore import QApplication, QUrl
import sys

class Example:

    def __init__(self):

        self.doRequest()

    def doRequest(self):

        url = "http://www.something.com"
        req = QtNetwork.QNetworkRequest(QUrl(url))

        self.nam = QtNetwork.QNetworkAccessManager()
        self.nam.finished.connect(self.handleResponse)
        self.nam.head(req)

    def handleResponse(self, reply):

        er = reply.error()

        if er == QtNetwork.QNetworkReply.NoError:

            for k, v in reply.rawHeaderPairs():
                print(str(k, 'UTF-8'), ": ", str(v, 'UTF-8'))

        else:
            print("Error occured: ", er)
            print(reply.errorString())

        QApplication.quit()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())
```

---

The example reads header fields from a network response.

```
self.nam.head(req)
```

The `head()` method is called.

```
for k, v in reply.rawHeaderPairs():
    print(str(k, 'UTF-8'), ": ", str(v, 'UTF-8'))
```

In this for loop, we go through the key/value pairs of the header data. The pairs are returned with `rawHeaderPairs()`.

### 8.3.4 Authentication

The `authenticationRequired` signal is emitted whenever a final server requests authentication before it delivers the requested contents.

Listing 8.7: Authentication

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we show how to authenticate
to a web page.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtCore, QtGui, QtNetwork
import sys, json


class Example:

    def __init__(self):

        self.doRequest()

    def doRequest(self):

        self.auth = 0

        url = "https://httpbin.org/basic-auth/user7/passwd7"
        req = QtNetwork.QNetworkRequest(QtCore.QUrl(url))

        self.nam = QtNetwork.QNetworkAccessManager()
        self.nam.authenticationRequired.connect(self.authenticate)
        self.nam.finished.connect(self.handleResponse)
        self.nam.get(req)

    def authenticate(self, reply, auth):

        print("Authenticating")

        self.auth += 1

        if self.auth >= 3:
            reply.abort()

        auth.setUser("user7")
        auth.setPassword("passwd7")

    def handleResponse(self, reply):

        er = reply.error()
```

```

        if er == QtNetwork.QNetworkReply.NoError:

            bytes_string = reply.readAll()

            data = json.loads(str(bytes_string, 'utf-8'))

            print('Authenticated: {0}'.format(
                data['authenticated']))
            print('User: {0}'.format(data['user']))

            print()

        else:
            print("Error occurred: ", er)
            print(reply.errorString())

    QtCore.QCoreApplication.quit()

app = QtCore.QCoreApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

In the example we use the <https://httpbin.org> website to show how authentication is done with `QNetworkAccessManager`.

```
self.nam.authenticationRequired.connect(self.authenticate)
```

We connect the `authenticationRequired` signal to the `authenticate()` method.

```

def authenticate(self, reply, auth):

    print("Authenticating")
    ...

```

The third parameter of the `authenticate()` method is the `QAuthenticator`, which is used to pass the required authentication information.

```

self.auth += 1

if self.auth >= 3:
    reply.abort()

```

The `QNetworkAccessManager` keeps emitting the `authenticationRequired` signal if the authentication fails. We abort the process after three failed attempts.

```

auth.setUser("user7")
auth.setPassword("passwd7")

```

We set the user and the password to the `QAuthenticator`.

```

bytes_string = reply.readAll()

data = json.loads(str(bytes_string, 'utf-8'))

print('Authenticated: {0}'.format(
    data['authenticated']))
print('User: {0}'.format(data['user']))

```

The <https://httpbin.org> responds with JSON data, which contains the user name and a boolean value indicating authentication success.

### 8.3.5 Fetching a favicon

A favicon is a small icon associated with a particular website. In this section we are going to download a favicon from a website.

Listing 8.8: Fetching a favicon

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example we fetch a favicon from
a website.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtCore, QtGui, QtNetwork
import sys

class Example:

    def __init__(self):

        self.doRequest()

    def doRequest(self):

        url = "http://www.google.com/favicon.ico"
        req = QtNetwork.QNetworkRequest(QtCore.QUrl(url))

        self.nam = QtNetwork.QNetworkAccessManager()
        self.nam.finished.connect(self.handleResponse)
        self.nam.get(req)

    def handleResponse(self, reply):

        er = reply.error()

        if er == QtNetwork.QNetworkReply.NoError:

            data = reply.readAll()
            self.saveFile(data)

        else:
            print("Error occurred: ", er)
            print(reply.errorString())

        QtCore.QCoreApplication.quit()

    def saveFile(self, data):

        f = open('favicon.ico', 'wb')
```

```

        with f:

            f.write(data)

app = QtCore.QCoreApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

The code example downloads a Google's favicon.

```
self.nam.get(req)
```

We download the icon with the `get()` method.

```
data = reply.readAll()
self.saveFile(data)
```

In the `handleResponse()` method we read the data and save it to the file.

```
def saveFile(self, data):

    f = open('favicon.ico', 'wb')

    with f:

        f.write(data)

```

The image data is saved on the disk in the `saveFile()` method.

## 8.4 Blocking and non-blocking requests

In GUI applications it is important that they remain responsive. Network requests take some time to finish. Therefore, most methods in `QtNetwork` are asynchronous or non-blocking. Asynchronous methods return immediately and do not wait for the request to be finished.

In console and multithreaded applications it is possible to use blocking or synchronous API. Synchronous API waits for the request and blocks the event loop.

---

Listing 8.9: Blocking request

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This is an example of a blocking
network program.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtNetwork
from PyQt5.QtCore import QObject, QTimer, QCoreApplication

```

```

import sys

class Example(QObject):

    def __init__(self):
        super().__init__()

        QTimer.singleShot(3000, self.openConnection)
        QTimer.singleShot(4000, self.doSomeWork)

    def openConnection(self):

        tcpSocket = QtNetwork.QTcpSocket()
        tcpSocket.connectToHost("www.something.com", 80)

        tcpSocket.write(b"GET / HTTP/1.1\r\n" \
                        b"Host: www.something.com\r\n" \
                        b"\r\n")

        while tcpSocket.waitForReadyRead():

            bytes_string = tcpSocket.readAll()
            print(str(bytes_string, 'utf-8'))

        if tcpSocket.error():
            print(tcpSocket.errorString())

        tcpSocket.close()

    def doSomeWork(self):

        print("Doing some work")
        QApplication.quit()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

A TCP connection is opened to port 80 of the something.com website. A simple GET request is sent through the socket. The `doSomeWork()` method is executed after the request is finished.

```

class Example(QObject):

    def __init__(self):
        super().__init__()

        QTimer.singleShot(3000, self.openConnection)
        QTimer.singleShot(4000, self.doSomeWork)

```

Synchronous API blocks the event loop. Therefore, we have to create some delay before calling the `openConnection()` method. This way we let the `Example` object be created.

```

def openConnection(self):

```

```

tcpSocket = QtNetwork.QTcpSocket()
tcpSocket.connectToHost("wikipedia.org", 80)
...

```

A connection to the TCP port 80 of the `www.something.com` website is created. The port is used by Hypertext Transfer Protocol (HTTP).

```

tcpSocket.write(b"GET / HTTP/1.1\r\n" \
                b"Host: www.something.com\r\n" \
                b"\r\n")

```

With the `write()` method of the socket, we create an HTTP request. The request has a specific syntax documented by the RFC 2616 standard. An HTTP request consists of a request line, headers, an empty line and an optional message body. The request line provides a command to be executed. We use the GET command to retrieve a root page of the provided website. The request line and the headers end with the carriage return and new line characters. The empty line must consist of only carriage return and new line characters.

```

while tcpSocket.waitForReadyRead():

    bytes_string = tcpSocket.readAll()
    print(str(bytes_string, 'utf-8'))

```

Data arrives in portions. The `readyRead` signal is emitted every time a new chunk of data has arrived. The `waitForReadyRead()` blocks until the new data is available. After some time it will timeout. The default timeout is 30000 milliseconds.

```

if tcpSocket.error():
    print(tcpSocket.errorString())

```

In case of an error, we print the error message.

```

tcpSocket.close()

```

In the end, we close the socket.

```

$ ./blocking.py
HTTP/1.1 200 OK
Date: Mon, 04 Sep 2017 10:33:11 GMT
Server: Apache/2.4.12 (FreeBSD) OpenSSL/1.0.11-freelsd
       mod_fastcgi/mod_fastcgi-SNAP-0910052141
Last-Modified: Mon, 25 Oct 1999 15:36:02 GMT
ETag: "4d-357b661867c80"
Accept-Ranges: bytes
Content-Length: 77
Vary: Accept-Encoding
Content-Type: text/html

<html><head><title>Something.</title></head>
<body>Something.</body>
</html>

```

```

The remote host closed the connection
Doing some work

```

We get a web page from `www.something.com`. Note that the `doSomeWork()` method is executed after the web page is read.

The following script is an example of a non-blocking network program. A



root page is retrieved from [www.something.com](http://www.something.com).

Listing 8.10: Non-blocking request

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This is an example of a non-blocking
network program.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtNetwork
from PyQt5.QtCore import QObject, QThread, QCoreApplication
import sys, time

class Worker(QThread):

    def __init__(self):
        super(Worker, self).__init__()

        print("Init Thread")

    def run(self):

        print("Thread started")

        time.sleep(10)

        print("Thread ended")

class Example(QObject):

    def __init__(self):
        super().__init__()

        self.openConnection()
        self.doSomeWork()

    def openConnection(self):

        self.tcpSocket = QtNetwork.QTcpSocket()
        self.tcpSocket.readyRead.connect(self.readData)
        self.tcpSocket.error.connect(self.onError)

        self.tcpSocket.connectToHost("www.something.com", 80)

        self.tcpSocket.write(b"GET / HTTP/1.1\r\n" \
                             b"Host: www.something.com\r\n" \
                             b"\r\n")

    def doSomeWork(self):
```

```

        self.wrk = Worker()
        self.wrk.finished.connect(QCoreApplication.quit)
        self.wrk.start()

    def onError(self, e):

        t = QtNetwork.QAbstractSocket.RemoteHostClosedError

        if e == t:
            print("closing connection")

        else:
            print("error")
            print(self.tcp.errorString())

    def readData(self):

        while not self.tcpSocket.atEnd():

            bytes_string = self.tcpSocket.readAll()
            print(str(bytes_string, 'utf-8'))

            self.tcpSocket.close()

app = QCoreApplication([])
ex = Example()
sys.exit(app.exec_())

```

The example uses asynchronous API to download a page. A separate thread is created to demonstrate a parallel task.

```

class Example(QObject):

    def __init__(self):
        super().__init__()

        self.openConnection()
        self.doSomeWork()

```

Asynchronous APIs do not block the program. Therefore, we can execute the `openConnection()` method right away.

```

def openConnection(self):

    self.tcpSocket = QtNetwork.QTcpSocket()
    self.tcpSocket.readyRead.connect(self.readData)
    self.tcpSocket.error.connect(self.onError)

    self.tcpSocket.connectToHost("www.something.com", 80)

    self.tcpSocket.write(b"GET / HTTP/1.1\r\n" \
                        b"Host: www.something.com\r\n" \
                        b"\r\n")

```

We connect to `readyRead` and `error` signals, open a connection to the HTTP port of the `www.something.com`, and get the root page. The control is immediately returned to the program. The two methods are executed when their signals are emitted. In the meantime, the `doSomeWork()` method can be launched.

```
def doSomeWork(self):

    self.wrk = Worker()
    self.wrk.finished.connect(QCoreApplication.quit)
    self.wrk.start()
```

The `doSomeWork()` method creates a thread. The application is closed when the thread ends.

```
def readData(self):

    while not self.tcpSocket.atEnd():

        bytes_string = self.tcpSocket.readAll()
        print(str(bytes_string, 'utf-8'))

    self.tcpSocket.close()
```

The `readyRead` signal launches the `readData()` method. The method reads all data received.

```
$ ./nonblocking.py
Init Thread
Thread started
HTTP/1.1 200 OK
Date: Mon, 04 Sep 2017 10:54:19 GMT
Server: Apache/2.4.12 (FreeBSD) OpenSSL/1.0.1l-freebsd
       mod_fastcgi/mod_fastcgi-SNAP-0910052141
Last-Modified: Mon, 25 Oct 1999 15:36:02 GMT
ETag: "4d-357b661867c80"
Accept-Ranges: bytes
Content-Length: 77
Vary: Accept-Encoding
Content-Type: text/html

<html><head><title>Something.</title></head>
<body>Something.</body>
</html>

closing connection
Thread ended
```

The output shows two jobs running in parallel. Opening connection to the web page does not block a task that is run in a thread.

## 8.5 Echo service

An echo service is an internet protocol defined in RFC 862. It is a useful debugging and measurement tool. The service simply sends any data received back to the source.

The `QTcpServer` class provides a TCP-based server. The class accepts incoming TCP connections on a specified port. We have chosen a custom 6001 port because the original port (7) for the echo service needs root privileges. The `listen()` method will have the server listen for incoming connections. The `newConnection()` signal is emitted each time a client connects to the server.

---

Listing 8.11: Echo server

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This is an example of an echo server.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtNetwork
from PyQt5.QtCore import QApplication
import sys, signal

class EchoSocket(QtNetwork.QTcpSocket):

    def __init__(self, p):
        super().__init__()

        self.readyRead.connect(self.readClient)
        self.disconnected.connect(self.discardClient)

        self.error.connect(self.onError)

    def readClient(self):

        msg = self.readLine()
        print(msg)
        self.write(msg)

    def discardClient(self):

        self.deleteLater()

    def onError(self, e):

        t = QtNetwork.QAbstractSocket.RemoteHostClosedError

        if e == t:
            print("closing connection")

        else:
            print("error")
            print(self.tcp.errorString())

class EchoServer(QtNetwork.QTcpServer):

    def __init__(self, port):
        super().__init__()

        print("Starting Echo Server")
        print("Listening on port:", port)
        self.listen(QtNetwork.QHostAddress.LocalHost, port)

```

```

def incomingConnection(self, socket):

    self.es = EchoSocket(self)
    self.es.setSocketDescriptor(socket)

signal.signal(signal.SIGINT, signal.SIG_DFL)

app = QApplication([])
echo = EchoServer(6001)
sys.exit(app.exec_())

```

This is an example of a echo server. The server listens on the 6001 port for incoming requests. It sends all data back to the client. It can be terminated with the Ctrl+C key combination.

```

class EchoSocket(QtNetwork.QTcpSocket):

    def __init__(self, p):
        super().__init__()

        self.readyRead.connect(self.readClient)
        self.disconnected.connect(self.discardClient)

        self.error.connect(self.onError)

```

For each connection, an `EchoSocket` is created.

```

def readClient(self):

    msg = self.readLine()
    print(msg)
    self.write(msg)

```

The `readLine()` method reads the data from the client. We print the data to the console. We send the data back to the client utilizing the `write()` method.

```

def discardClient(self):

    self.deleteLater()

```

A server may potentially receive huge amounts of requests. For each request, a new instance of a `QTcpSocket` is created. To avoid large consumption of memory, the objects are explicitly deleted with the `deleteLater()` method. It schedules the `EchoSocket` object for deletion. The object will be deleted when control returns to the event loop.

```

self.listen(QtNetwork.QHostAddress.LocalHost, port)

```

We tell the server to listen for incoming connections on localhost and on a specified port.

```

def incomingConnection(self, socket):

    self.es = EchoSocket(self)
    self.es.setSocketDescriptor(socket)

```

The `incomingConnection()` is called when a new connection is available. A new `EchoSocket` object is created. The `setSocketDescriptor()` initializes the native

socket descriptor. A socket descriptor is a unique number that identifies a socket and a TCP/IP connection.

The following program connects to the echo server and sends a message to it.

---

Listing 8.12: Echo client

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This is an example of an echo client.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtNetwork
from PyQt5.QtCore import QObject, QCoreApplication
import sys


class EchoClient(QObject):

    def __init__(self):
        super().__init__()

        self.getMessage()
        self.openConnection()

    def getMessage(self):

        args = sys.argv

        if len(args) == 2:

            self.msg = bytearray(args[1], 'UTF-8')

        else:
            print("Usage: echo_client.py message")
            sys.exit(1)

    def openConnection(self):

        self.tcpSocket = QtNetwork.QTcpSocket()
        self.tcpSocket.readyRead.connect(self.readData)
        self.tcpSocket.error.connect(self.onError)

        self.tcpSocket.connectToHost("localhost", 6001)

        self.tcpSocket.write(self.msg)

    def onError(self, e):
```

```

t = QtNetwork.QAbstractSocket.RemoteHostClosedError

if e == t:
    print("closing connection")

else:
    print("error")
    print(self.tcp.errorString())

def readData(self):

    while not self.tcpSocket.atEnd():

        bytes_string = self.tcpSocket.readAll()
        print(str(bytes_string, 'utf-8'))

        self.tcpSocket.close()
        QApplication.quit()

app = QApplication([])
ec = EchoClient()
sys.exit(app.exec_())

```

---

The program takes one parameter. It is the message to be sent to the Echo server.

```

def openConnection(self):

    self.tcpSocket = QtNetwork.QTcpSocket()
    self.tcpSocket.readyRead.connect(self.readData)
    self.tcpSocket.error.connect(self.onError)

    self.tcpSocket.connectToHost("localhost", 6001)

    self.tcpSocket.write(self.msg)

```

We open a connection to the localhost on port 6001. A message is written to the socket.

```

def readData(self):

    while not self.tcpSocket.atEnd():

        bytes_string = self.tcpSocket.readAll()
        print(str(bytes_string, 'utf-8'))

        self.tcpSocket.close()
        QtCore.QCoreApplication.quit()

```

We read the response from the server and close the socket. The application is terminated.

## 8.6 Web server

In this section, we create a simple skeleton of a Web server. The server is based on the `QTcpServer` and listens on the 8080 port.

The Hypertext Transfer Protocol (HTTP) is an application protocol for dis-

tributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity metainformation, and possible entity-body content.<sup>1</sup>

The current HTTP/1.1 protocol is defined in the RFC 2616 document.

Listing 8.13: Simple web server

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This is a simple skeleton of an web server.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtNetwork
from PyQt5.QtCore import QObject, QCoreApplication, QTextStream

class HttpSocket(QtNetwork.QTcpSocket):

    def __init__(self, p):
        super().__init__()

        self.readyRead.connect(self.readClient)
        self.disconnected.connect(self.discardClient)

        self.error.connect(self.onError)

    def readClient(self):

        cmd = self.readLine().split(' ')[0]

        if cmd.toUpper() == 'HEAD':
            self.do_head()

        elif cmd.toUpper() == 'GET':
            self.do_get()

        else:
            self.do_error()

    def do_head(self):

        os = QTextStream(self)
```

---

<sup>1</sup><http://www.ietf.org/rfc/rfc2616.txt>



```

        os.setAutoDetectUnicode(True)
        os << 'HTTP/1.1 200 OK\r\n' \
            + 'Content-Type: text/plain; charset="utf-8"\r\n' \
            + 'Server: HTTP skeleton server\r\n' \
            + '\r\n'
        self.close()

    def do_get(self):

        os = QTextStream(self)

        os.setAutoDetectUnicode(True)
        os << 'HTTP/1.1 200 OK\r\n' \
            + 'Content-Type: text/plain; charset="utf-8"\r\n' \
            + 'Server: HTTP skeleton server\r\n' \
            + '\r\n' \
            + 'Simple web page\n'
        self.close()

    def do_error(self):

        os = QTextStream(self)
        os.setAutoDetectUnicode(True)
        os << "HTTP/1.1 400 Bad Request\r\n" \
            + "Content-Type: text/plain; charset=\"utf-8\"\r\n" \
            + "\r\n" \
            + 'The request could not be understood by the\n' \
            + 'server due to malformed syntax.\n'
        self.close()

    def discardClient(self):

        self.deleteLater()
        print("Connection closed")

    def onError(self, error):

        print("Error: ", error)

class HttpServer(QtNetwork.QTcpServer):

    def __init__(self, port):
        super().__init__()

        print("Starting HTTP Server")
        print("Listening on port:", port)
        self.listen(QtNetwork.QHostAddress.Any, port)

    def incomingConnection(self, socket):

        self.hs = HttpSocket(self)
        self.hs.setSocketDescriptor(socket)

import sys, signal

```

```

signal.signal(signal.SIGINT, signal.SIG_DFL)

app = QApplication([])
http = HttpServer(8080)
sys.exit(app.exec_())

```

---

This program will respond to two HTTP commands: GET and HEAD.

```

class HttpSocket(Qtnetwork.QTcpSocket):

    def __init__(self, p):
        super().__init__(p)
        ...

```

Each request will be processed by an `HttpSocket`.

```

def readClient(self):

    cmd = self.readLine().split(' ')[0]

    if cmd.toUpper() == 'HEAD':
        self.do_head()

    elif cmd.toUpper() == 'GET':
        self.do_get()

    else:
        self.do_error()

```

A client request is parsed by the `readClient()` method. For simplicity reasons, the server recognizes only HEAD and GET commands.

```

def do_head(self):

    os = QTextStream(self)

    os.setAutoDetectUnicode(True)
    os << 'HTTP/1.1 200 OK\r\n' \
        + 'Content-Type: text/plain; charset="utf-8"\r\n' \
        + 'Server: HTTP skeleton server\r\n' \
        + '\r\n'
    self.close()

```

This method responds to a HEAD HTTP command. The response has a precise syntax. For example, carriage return and new line characters are used to separate header field values. Unlike a GET command, a HEAD command does not contain a message-body in the response.

```

def do_get(self):

    os = QTextStream(self)

    os.setAutoDetectUnicode(True)
    os << 'HTTP/1.1 200 OK\r\n' \
        + 'Content-Type: text/plain; charset="utf-8"\r\n' \
        + 'Server: HTTP skeleton server\r\n' \
        + '\r\n' \
        + 'Simple web page\n'
    self.close()

```

This method returns a response to a GET command. It also contains a simple

message body. The message body has a plain textual format.

```
$ ./httpd.py
Starting HTTP Server
Listening on port: 8080
$ curl localhost:8080
Simple web page
$ curl -I localhost:8080
HTTP/1.1 200 OK
Content-Type: text/plain; charset="utf-8"
Server: HTTP skeleton server
```

We use the `curl` tool to test the application.

## 8.7 Connecting to an SMTP server

SMTP stands for a Simple Mail Transfer Protocol. It is used to transfer mail reliably and efficiently. SMTP uses TCP port 25. SMTP Message submission (SMTP-MSA) is a variant of the SMTP protocol. It uses port 587. The latter protocol is used by Gmail.

---

Listing 8.14: Connecting to an SMTP server

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we send an EHLO command
to an SMTP server.

Author: Jan Bodnar
Website: zetcode.com
Last edited: September 2017
'''

from PyQt5 import QtNetwork
from PyQt5.QtCore import QObject, QCoreApplication
import sys

class Example(QObject):

    states = ("unconnected", "looking up host", "connecting",
             "connected", "bound", "listening", "closing")

    def __init__(self):
        super().__init__()

        self.initExample()

    def initExample(self):

        self.tcp = QtNetwork.QTcpSocket(self)

        self.tcp.readyRead.connect(self.readData)
        self.tcp.stateChanged.connect(self.stateChanged)
        self.tcp.error.connect(self.onError)
```

```

        self.tcp.connectToHost("smtp.gmail.com", 587)
        self.tcp.write(b"EHLO smtp.gmail.com\r\n")
        self.tcp.write(b"QUIT\r\n")

    def readData(self):

        while not self.tcp.atEnd():

            bytes_string = self.tcp.readAll()
            print(str(bytes_string, 'utf-8'))

    def stateChanged(self, state):

        print(self.states[state])

    def onError(self, error):

        t = QtNetwork.QAbstractSocket.RemoteHostClosedError

        if error == t:
            print("closing connection")

        else:
            print("error")
            print(self.tcp.errorString())

        self.tcp.close()
        QApplication.quit()

app = QApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

The `QTcpSocket` class is used to connect to a Google's email server. An email is sent via a sequence of SMTP commands. Our example sends two basic commands to the Google's email server. The server responds with a greeting and closes the connection. The example also prints the connection's state.

```

def initExample(self):

    self.tcp = QtNetwork.QTcpSocket(self)

    self.tcp.readyRead.connect(self.readData)
    self.tcp.stateChanged.connect(self.stateChanged)
    self.tcp.error.connect(self.onError)
    ...

```

We create an instance of the `QTcpSocket` class and three connections to signals.

```

self.tcp.connectToHost("smtp.gmail.com", 587)
self.tcp.write(b"EHLO smtp.gmail.com\r\n")
self.tcp.write(b"QUIT\r\n")

```

A connection to smtp.gmail.com server on port 587 is established. We issue EHLO and QUIT commands.

```

def readData(self):

```

```

while not self.tcp.atEnd():

    bytes_string = self.tcp.readAll()
    print(str(bytes_string, 'utf-8'))

```

The server's response to the two commands is read and printed to the console.

```

def stateChanged(self, state):

    print self.states[state]

```

We also print the current connection's state.

```

$ ./gmail.py
looking up host
connecting
connected
220 smtp.gmail.com ESMTP f5sm3671465edb.90 - gsmt
250-smtp.gmail.com at your service,
    [2a02:ab04:1b8:e500:b09e:3fe5:ea6e:d68]
250-SIZE 35882577
250-8BITMIME
250-STARTTLS
250-ENHANCEDSTATUSCODES
250-PIPELINING
250-CHUNKING
250 SMTPUTF8

221 2.0.0 closing connection f5sm3671465edb.90 - gsmt

closing connection
closing
unconnected

```

This is the output of the program. We see various connection states and the response of the Google's email server.

## 8.8 Weather conditions

Yahoo has a free service that provides weather information for a specific location. The location is identified by a WOEID code. The WOEID (Where on Earth Identifier) is a unique 32 bit reference identifier assigned by Yahoo to identify any feature on Earth. WOEIDs can be looked up at [www.woeidlookup.com](http://www.woeidlookup.com).

Yahoo uses the YQL (Yahoo Query Language) to query, filter, and combine data across the web through a single interface. There are two parameters possible: the `woeid` parameter for the WOEID and the `u` parameter for degrees units (Fahrenheit or Celsius). The `woeid` parameter is mandatory.

---

Listing 8.15: Weather conditions

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

In this example, we get the weather conditions
for a particular city.

```

*Author: Jan Bodnar*  
*Website: zetcode.com*  
*Last edited: September 2017*  
,,

```
from PyQt5 import QtNetwork
from PyQt5.QtCore import (QObject, QUrl, QCoreApplication,
    QUrlQuery, QXmlStreamReader)
from urllib.request import urlopen
from urllib.parse import urlencode
import sys

class Example(QObject):

    def __init__(self):
        super().__init__()

        self.buildUrl()
        self.checkWeather()

    def buildUrl(self):

        args = sys.argv

        if len(args) == 2:

            wid = args[1]

            baseurl = "https://query.yahooapis.com/v1/public/yql?"
            yql_query = "select * from weather.forecast where " \
                + "woeid={0} and u='c'".format(wid)

            yql_url = baseurl + urlencode({'q':yql_query})

            self.url = QUrl(yql_url)

        else:
            print("Usage: weather.py WOEID")
            sys.exit(1)

    def checkWeather(self):

        self.nam = QtNetwork.QNetworkAccessManager()
        self.nam.finished.connect(self.handleResponse)

        req = QtNetwork.QNetworkRequest(self.url)
        self.nam.get(req)

    def handleResponse(self, reply):

        er = reply.error()

        if er == QtNetwork.QNetworkReply.NoError:

            data = reply.readAll()

            r = QXmlStreamReader(str(data, 'UTF-8'))
```

```

while not r.atEnd():

    r.readNext()

    if (r.qualifiedName() == "yweather:location"):

        if r.isStartElement():

            atr = r.attributes()
            print("City:", atr.value("city"))
            print("Country:", atr.value("country"))

    if (r.qualifiedName() == "yweather:condition"):

        if r.isStartElement():

            atr = r.attributes()
            print("Date:", atr.value("date"))
            print("Condition:", atr.value("text"))
            print("Temperature:", \
                  atr.value("temp"), "deg C")

    if r.hasError():
        print("Error reading feed")

    else:
        print("Error occurred: ", er)
        print(er)
        print(reply.errorString())

QtCoreApplication.quit()

app = QtCoreApplication([])
ex = Example()
sys.exit(app.exec_())

```

---

The example prints basic weather information for a specified WOEID.

```

baseurl = "https://query.yahooapis.com/v1/public/yql?"
yql_query = "select * from weather.forecast where " \
            + "woeid={0} and u='c'".format(wid)

```

We have the base URL for the web service and the YQL query.

```

def checkWeather(self):

    self.nam = QtNetwork.QNetworkAccessManager()
    self.nam.finished.connect(self.handleResponse)

    req = QtNetwork.QNetworkRequest(self.url)
    self.nam.get(req)

```

With the `QNetworkAccessManager`, we create a GET request to the URL.

```

r = QDomStreamReader(str(data, 'UTF-8'))

```

The web service responds in XML; we use `QXmlStreamReader` to parse the XML data.

```

if (r.qualifiedName() == "yweather:location"):

```

```

if r.isStartElement():

    atr = r.attributes()
    print("City:", atr.value("city"))
    print("Country:", atr.value("country"))

```

The attributes of the <yweather:location> tag contain the city and country values.

```

if (r.qualifiedName() == "yweather:condition"):

    if r.isStartElement():

        atr = r.attributes()
        print("Date:", atr.value("date"))
        print("Condition:", atr.value("text"))
        print("Temperature:", \
            atr.value("temp"), "deg C")

```

The attributes of the <yweather:condition> tag contain the weather conditions information.

```

$ ./weather.py 820323
City: Kosice
Country: Slovakia
Date: Mon, 04 Sep 2017 04:00 PM CEST
Condition: Showers
Temperature: 15 deg C

```

The example is run with a WOEID for Košice.



## Chapter 9

# Games

In this chapter we are going to create three computer games: Snake, Sokoban, and Minesweeper.

### 9.1 Snake

Snake is an older classic video game. It was first created in late 70s. Later it was brought to PCs. In this game the player controls a snake. The objective is to eat as many apples as possible. Each time the snake eats an apple, its body grows. The snake must avoid the walls and its own body.

Listing 9.1: Snake game

---

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This is a simple Snake game clone.

Author: Jan Bodnar
Website: zetcode.com
Last edited: October 2017
'''

from PyQt5.QtWidgets import QWidget, QMainWindow, QApplication
from PyQt5.QtGui import (QPainter, QImage, QColor,
                          QFont, QFontMetrics)
from PyQt5.QtCore import Qt, QBasicTimer, QPoint

import random
import sys

WIDTH = 300
HEIGHT = 270
DOT_SIZE = 10
ALL_DOTS = WIDTH * HEIGHT // (DOT_SIZE * DOT_SIZE)
RAND_POS = 26
DELAY = 140
```

```

x = [0] * ALL_DOTS
y = [0] * ALL_DOTS

class Board(QWidget):
    def __init__(self):
        super().__init__()

        self.setFocusPolicy(Qt.StrongFocus)

        self.initGame()

    def timerEvent(self, event):
        if self.inGame:
            self.checkApple()
            self.checkCollision()
            self.move()
        else:
            self.timer.stop()

        self.repaint()

    def initGame(self):
        self.left = False
        self.right = True
        self.up = False
        self.down = False
        self.inGame = True
        self.dots = 3

        for i in range(self.dots):
            x[i] = 50 - i * 10
            y[i] = 50

        try:
            self.ball = QImage("dot.png")
            self.apple = QImage("apple.png")
            self.head = QImage("head.png")

        except Exception as e:
            print(e.message)
            sys.exit(1)

        self.locateApple()
        self.timer = QTimer()
        self.timer.start(DELAY, self)

    def paintEvent(self, event):
        super().paintEvent(event)

        painter = QPainter()
        painter.begin(self)

        painter.setBrush(QColor(0, 0, 0, 255))

```

```

        painter.drawRect(0, 0, self.width(), self.height())

    if self.inGame:
        self.drawObjects(painter)
    else:
        self.gameOver(painter)

    painter.end()

def drawObjects(self, painter):

    painter.drawImage(self.apple_x, self.apple_y, self.apple)

    for z in range(self.dots):
        if z == 0:
            painter.drawImage(x[z], y[z], self.head)
        else:
            painter.drawImage(x[z], y[z], self.ball)

def gameOver(self, painter):

    msg = "Game Over"
    small = QFont("Helvetica", 12, QFont.Bold)
    metr = QFontMetrics(small)

    textWidth = metr.width(msg)
    h = self.height()
    w = self.width()

    painter.setPen(QColor(255, 255, 255))
    painter.setFont(small)
    painter.translate(QPoint(w/2, h/2))
    painter.drawText(-textWidth/2, 0, msg)

def checkApple(self):

    if x[0] == self.apple_x and y[0] == self.apple_y:

        self.dots = self.dots + 1
        self.locateApple()

def move(self):

    z = self.dots

    while z > 0:

        x[z] = x[(z - 1)]
        y[z] = y[(z - 1)]
        z = z - 1

    if self.left:
        x[0] -= DOT_SIZE

    if self.right:
        x[0] += DOT_SIZE

    if self.up:

```

```

        y[0] -= DOT_SIZE

    if self.down:
        y[0] += DOT_SIZE

def checkCollision(self):

    z = self.dots

    while z > 0:
        if z > 4 and x[0] == x[z] and y[0] == y[z]:
            self.inGame = False
            z = z - 1

    if y[0] > HEIGHT - DOT_SIZE:
        self.inGame = False

    if y[0] < 0:
        self.inGame = False

    if x[0] > WIDTH - DOT_SIZE:
        self.inGame = False

    if x[0] < 0:
        self.inGame = False

def locateApple(self):

    r = random.randint(0, RAND_POS)
    self.apple_x = r * DOT_SIZE
    r = random.randint(0, RAND_POS)
    self.apple_y = r * DOT_SIZE

def keyPressEvent(self, event):

    key = event.key()

    if key == Qt.Key_Left and not self.right:

        self.left = True
        self.up = False
        self.down = False

    if key == Qt.Key_Right and not self.left:

        self.right = True
        self.up = False
        self.down = False

    if key == Qt.Key_Up and not self.down:

        self.up = True
        self.right = False
        self.left = False

    if key == Qt.Key_Down and not self.up:

        self.down = True
        self.right = False

```

```

        self.left = False

class Snake(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        self.setWindowTitle("Snake")
        self.resize(WIDTH, HEIGHT)
        self.setCentralWidget(Board())

        self.move(300, 300)
        self.show()

def main():

    app = QApplication([])
    snake = Snake()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

---

The size of each of the joints of a snake is 10 px. The snake is controlled with the cursor keys. Initially, the snake has three joints. The game starts immediately. When the game is finished, we display “Game Over” message in the center of the window.

```

WIDTH = 300
HEIGHT = 270
DOT_SIZE = 10
ALL_DOTS = WIDTH * HEIGHT // (DOT_SIZE * DOT_SIZE)
RAND_POS = 26
DELAY = 140

```

The `WIDTH` and `HEIGHT` constants determine the size of the board. The `DOT_SIZE` is the size of the apple and the dot of the snake. The `ALL_DOTS` constant defines the maximum number of possible dots on the board. The `RAND_POS` constant is used to calculate a random position of an apple. The `DELAY` constant determines the speed of the game.

```

x = [0] * ALL_DOTS
y = [0] * ALL_DOTS

```

These two lists store `x` and `y` coordinates of all possible joints of a snake.

The `initGame()` method initiates the game.

```

self.left = False
self.right = True
self.up = False
self.down = False
self.inGame = True
self.dots = 3

```

We initiate variables that we use in the game.

```
for i in range(self.dots):  
    x[i] = 50 - i * 10  
    y[i] = 50
```

We give the snake joints initial coordinates. It always starts from the same position.

```
try:  
    self.ball = QImage("dot.png")  
    self.apple = QImage("apple.png")  
    self.head = QImage("head.png")  
  
except Exception as e:  
    print(e.message)  
    sys.exit(1)
```

We load the images for the game.

```
self.locateApple()
```

The apple goes to an initial random position.

```
self.timer = QTimer()  
self.timer.start(DELAY, self)
```

We create a timer object and start it. The timer calls the `timerEvent()` method every DELAY milliseconds.

```
def timerEvent(self, event):  
    if self.inGame:  
        self.checkApple()  
        self.checkCollision()  
        self.move()  
    else:  
        self.timer.stop()  
  
    self.repaint()
```

Every 140 ms, the `timerEvent()` method is called. If we are in the game, we call three methods that build the logic of the game; otherwise we stop the timer.

```
if self.inGame:  
    self.drawObjects(painter)  
else:  
    self.gameOver(painter)
```

Inside the `paintEvent()` method, we check the `inGame` variable. If it is true, we draw our objects: the apple and the snake joints; otherwise we display “Game Over” text.

```
def drawObjects(self, painter):  
    painter.drawImage(self.apple_x,  
                      self.apple_y, self.apple)  
  
    for z in range(self.dots):  
        if z == 0:  
            painter.drawImage(x[z], y[z], self.head)
```

```

        else:
            painter.drawImage(x[z], y[z], self.ball)

```

The `drawObjects()` method draws the apple and the joints of the snake. The first joint of a snake is its head, which is represented by a red circle.

```

def checkApple(self):

    if x[0] == self.apple_x and y[0] == self.apple_y:
        self.dots = self.dots + 1
        self.locateApple()

```

The `checkApple()` method checks if the snake has hit the apple object. If so, we add another snake joint and call the `locateApple()` method, which randomly places a new apple object.

In the `move()` method we have the key algorithm of the game. To understand it, look at how the snake is moving. You control the head of the snake. You can change its direction with the cursor keys. The rest of the joints move one position up the chain. The second joint moves where the first was, the third joint where the second was etc.

```

while z > 0:
    x[z] = x[(z - 1)]
    y[z] = y[(z - 1)]
    z = z - 1

```

This code moves the joints up the chain.

```

if self.left:
    x[0] -= DOT_SIZE

```

The head is moved to the left.

In the `checkCollision()` method, we determine if the snake has hit itself or one of the walls.

```

while z > 0:
    if z > 4 and x[0] == x[z] and y[0] == y[z]:
        self.inGame = False
    z = z - 1

```

The game is finished if the snake hits one of its joints with the head.

```

if y[0] > HEIGHT - DOT_SIZE:
    self.inGame = False

```

The game is finished if the snake hits the bottom of the Board.

The `locateApple()` method locates an apple randomly on the board.

```

r = random.randint(0, RAND_POS)

```

We get a random number from 0 to `RAND_POS-1`.

```

self.apple_x = r * DOT_SIZE
...
self.apple_y = r * DOT_SIZE

```

These lines set the `x` and `y` coordinates of the apple.

In the `keyPressEvent()` method of the `Board` class, we determine the keys that were pressed.

```

if key == Qt.Key_Left and not self.right:

    self.left = True
    self.up = False
    self.down = False

```

If we hit the left cursor key, we set `left` variable to `True`. This variable is used in the `move()` method to change coordinates of the snake object. Notice also that when the snake is heading to the right, we cannot turn immediately to the left.

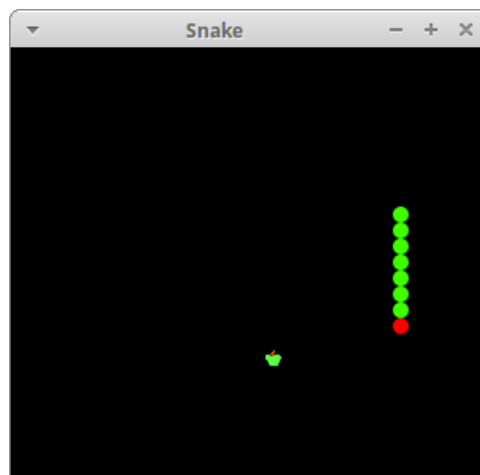


Figure 9.1: Snake game

## 9.2 Sokoban

Sokoban is another classic computer game. It was created in 1980 by Hiroyuki Imabayashi. Sokoban means a warehouse keeper in Japanese. The player pushes boxes around a maze. The objective is to place all boxes in designated locations.

We control the sokoban object with cursor keys. We can also press the R key to restart the level. When all bags are placed on the destination areas, the game is finished. We draw “Completed” string in the left upper corner of the window.

Listing 9.2: Sokoban game

---

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This is a simple Sokoban game clone.

Author: Jan Bodnar
Website: zetcode.com

```



*Last edited: September 2017*  
,,,

```
from PyQt5.QtWidgets import QWidget, QMainWindow, QApplication
from PyQt5.QtGui import QPainter, QImage, QColor
from PyQt5.QtCore import Qt
import os, sys
```

```
SPACE = 20
OFFSET = 30
```

```
LEFT_COLLISION = 1
RIGHT_COLLISION = 2
TOP_COLLISION = 3
BOTTOM_COLLISION = 4
```

```
level = """
#####
##   #
##$  #
#### $##
##  $ $ #
#### # ## # #####
##  # ## ##### ..#
## $ $      ..#
##### ## # @## ..#
##          #####
#####
"""
```

```
class Actor:

    def __init__(self, x, y):

        self._x = x
        self._y = y

    def getImage(self):
        return self.image

    def x(self):
        return self._x

    def y(self):
        return self._y

    def setX(self, x):
        self._x = x

    def setY(self, y):
        self._y = y

    def isLeftCollision(self, actor):

        if ((self.x() - SPACE) == actor.x()) and \
            (self.y() == actor.y()):
            return True
        else:
            return False
```

```

def isRightCollision(self, actor):

    if ((self.x() + SPACE) == actor.x()) and \
        (self.y() == actor.y()):
        return True
    else:
        return False

def isTopCollision(self, actor):

    if ((self.y() - SPACE) == actor.y()) and \
        (self.x() == actor.x()):
        return True
    else:
        return False

def isBottomCollision(self, actor):

    if ((self.y() + SPACE) == actor.y()) and \
        (self.x() == actor.x()):
        return True
    else:
        return False

class Baggage(Actor):

    def __init__(self, x, y):
        super().__init__(x, y)

        try:
            self.image = QImage("baggage.png")

        except Exception as e:

            print(e.message)
            sys.exit(1)

    def move(self, x, y):

        self._x += x
        self._y += y

class Player(Actor):

    def __init__(self, x, y):
        super().__init__(x, y)

        try:
            self.image = QImage("sokoban.png")

        except Exception as e:

            print(e.message)
            sys.exit(1)

    def move(self, x, y):

        self._x += x
        self._y += y

```

```

class Wall(Actor):

    def __init__(self, x, y):
        super().__init__(x, y)

        try:
            self.image = QImage("wall.png")

        except Exception as e:

            print(e.message)
            sys.exit(1)


class Area(Actor):

    def __init__(self, x, y):
        super().__init__(x, y)

        try:
            self.image = QImage("area.png")

        except Exception as e:

            print(e.message)
            sys.exit(1)


class Board(QWidget):

    def __init__(self):
        super().__init__()

        self.setFocusPolicy(Qt.StrongFocus)

        self.initBoard()

    def initBoard(self):

        self.world = []
        self.walls = []
        self.baggs = []
        self.areas = []

        self.w = 0
        self.h = 0

        self.completed = False
        self.soko = None

        self.initWorld()
        self.resize(self.w, self.h)

    def getWidth(self):
        return self.w

    def getHeight(self):

```

```

        return self.h

def initWorld(self):

    x = OFFSET
    y = OFFSET

    for i in range(len(level)):

        item = level[i]

        if item == '\n':

            y += SPACE

            if self.w < x:
                self.w = x

            x = OFFSET

        elif item == '#':

            w = Wall(x, y)
            self.walls.append(w)
            x += SPACE

        elif item == '$':

            b = Baggage(x, y)
            self.baggs.append(b)
            x += SPACE

        elif item == '.':

            a = Area(x, y)
            self.areas.append(a)
            x += SPACE

        elif item == '@':

            self.soko = Player(x, y)
            x += SPACE

        elif item == " ":
            x += SPACE

    self.h = y

def buildWorld(self, painter):

    painter.setBrush(QColor(250, 240, 170))
    painter.setPen(QColor(250, 240, 170))
    painter.drawRect(0, 0, self.width(), self.height())
    painter.setPen(QColor(0, 0, 0))

    world = self.walls + self.areas + \
            self.baggs + [self.soko]

    for i in range(len(world)):

```

```

        item = world[i]

        if isinstance(item, Player) or \
            isinstance(item, Baggage):
            painter.drawImage(item.x()+2, item.y()+2,
                              item.getImage())
        else:
            painter.drawImage(item.x(), item.y(),
                              item.getImage())

    if self.completed:
        painter.drawText(25, 20, "Completed")

def paintEvent(self, event):
    super().paintEvent(event)

    painter = QPainter()
    painter.begin(self)
    self.buildWorld(painter)
    painter.end()

def keyPressEvent(self, event):
    if self.completed:
        return

    key = event.key()

    if key == Qt.Key_Left:

        if self.checkWallCollision(self.soko,
                                    LEFT_COLLISION):
            return

        if self.checkBagCollision(LEFT_COLLISION):
            return

        self.soko.move(-SPACE, 0)

    elif key == Qt.Key_Right:

        if self.checkWallCollision(self.soko,
                                    RIGHT_COLLISION):
            return

        if self.checkBagCollision(RIGHT_COLLISION):
            return

        self.soko.move(SPACE, 0)

    elif key == Qt.Key_Up:

        if self.checkWallCollision(self.soko,
                                    TOP_COLLISION):
            return

        if self.checkBagCollision(TOP_COLLISION):
            return

```

```

        self.soko.move(0, -SPACE)

    elif key == Qt.Key_Down:

        if self.checkWallCollision(self.soko,
                                   BOTTOM_COLLISION):
            return

        if self.checkBagCollision(BOTTOM_COLLISION):
            return

        self.soko.move(0, SPACE)

    elif key == Qt.Key_R:
        self.restartLevel()

    self.repaint()

def checkWallCollision(self, actor, _type):

    if _type == LEFT_COLLISION:

        for i in range(len(self.walls)):
            wall = self.walls[i]
            if actor.isLeftCollision(wall):
                return True
        return False

    elif _type == RIGHT_COLLISION:

        for i in range(len(self.walls)):
            wall = self.walls[i]
            if actor.isRightCollision(wall):
                return True
        return False

    elif _type == TOP_COLLISION:

        for i in range(len(self.walls)):
            wall = self.walls[i]
            if actor.isTopCollision(wall):
                return True
        return False

    elif _type == BOTTOM_COLLISION:

        for i in range(len(self.walls)):
            wall = self.walls[i]
            if actor.isBottomCollision(wall):
                return True
        return False

def checkBagCollision(self, _type):

    if _type == LEFT_COLLISION:

        for i in range(len(self.baggs)):
            bag = self.baggs[i]
            if self.soko.isLeftCollision(bag):

```

```

        for i in range(len(self.baggs)):
            item = self.baggs[i]
            if not bag is item:
                if bag.isLeftCollision(item):
                    return True
            if self.checkWallCollision(bag,
                LEFT_COLLISION):
                return True
        bag.move(-SPACE, 0)
        self.isCompleted()

    return False

elif _type == RIGHT_COLLISION:

    for i in range(len(self.baggs)):
        bag = self.baggs[i]
        if self.soko.isRightCollision(bag):
            for i in range(len(self.baggs)):
                item = self.baggs[i]
                if not bag is item:
                    if bag.isRightCollision(item):
                        return True
            if self.checkWallCollision(bag,
                RIGHT_COLLISION):
                return True
        bag.move(SPACE, 0)
        self.isCompleted()

    return False

elif _type == TOP_COLLISION:

    for i in range(len(self.baggs)):
        bag = self.baggs[i]
        if self.soko.isTopCollision(bag):
            for i in range(len(self.baggs)):
                item = self.baggs[i]
                if not bag is item:
                    if bag.isTopCollision(item):
                        return True
            if self.checkWallCollision(bag,
                TOP_COLLISION):
                return True
        bag.move(0, -SPACE)
        self.isCompleted()

    return False

elif _type == BOTTOM_COLLISION:

    for i in range(len(self.baggs)):
        bag = self.baggs[i]
        if self.soko.isBottomCollision(bag):
            for i in range(len(self.baggs)):
                item = self.baggs[i]
                if not bag is item:
                    if bag.isBottomCollision(item):
                        return True
            if self.checkWallCollision(bag,
                BOTTOM_COLLISION):
                return True

```

```

        bag.move(0, SPACE)
        self.isCompleted()

    return False

def isCompleted(self):

    num = len(self.baggs)
    completed = 0

    for i in range(num):
        bag = self.baggs[i]
        for j in range(num):
            area = self.areas[j]
            if bag.x() == area.x() and \
                bag.y() == area.y():
                completed += 1

    if completed == num:
        self.completed = True
        self.repaint()

def restartLevel(self):

    del self.areas[:]
    del self.baggs[:]
    del self.walls[:]
    self.initWorld()

    if self.completed:
        self.completed = False

class Sokoban(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        self.setWindowTitle("Sokoban")
        board = Board()
        self.setCentralWidget(board)
        self.resize(board.getWidth() + OFFSET,
                    board.getHeight() + OFFSET)
        self.move(300, 300)
        self.show()

def main():

    app = QApplication([])
    sokoban = Sokoban()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

---

The game is simplified so that it is easier to understand. The game has one



level.

```
SPACE = 20
OFFSET = 30

LEFT_COLLISION = 1
RIGHT_COLLISION = 2
TOP_COLLISION = 3
BOTTOM_COLLISION = 4
```

The wall image's size is 20x20 px. This reflects the `SPACE` constant. The `OFFSET` is the distance between the borders of the window and the game world. There are four types of collisions. Each one is represented by a numerical constant.

```
level = """
#####
##   #
##$  #
#### $##
##  $ $ #
#### # ## # #####
##   # ## ##### ..#
##  $ $      ..#
##### ### #@## ..#
##           #####
#####
"""
```

This is the level of the game. Except for the space, there are four characters. The hash (#) stands for a wall. The dollar (\$) represents the box to move. The dot (.) character represents the place where we must move the box. Finally, the at character (@) is the sokoban.

```
class Actor:

    def __init__(self, x, y):

        self._x = x
        self._y = y
    ...
```

This is the constructor of the `Actor` class. This class is a base class for other classes in the game. It encapsulates the basic functionality of an object in the Sokoban game.

```
def x(self):
    return self._x

def y(self):
    return self._y

def setX(self, x):
    self._x = x

def setY(self, y):
    self._y = y
```

These methods set and get the coordinates of an object.

```
def isLeftCollision(self, actor):

    if ((self.x() - SPACE) == actor.x()) and \
```

```

        (self.y() == actor.y()):
            return True
    else:
        return False

```

The `isLeftCollision()` method checks if the object collides with another object on the left side.

```

class Player(Actor):

    def __init__(self, x, y):
        super().__init__(x, y)

    try:
        self.image = QImage("sokoban.png")

    except Exception as e:

        print(e.message)
        sys.exit(1)

    def move(self, x, y):

        self._x += x
        self._y += y

```

The `Player` class inherits functionality from the base `Actor` class. It loads an image which represents the sokoban object in the game. It has an additional `move()` method. The method is responsible for moving the object.

```

self.world = []
self.walls = []
self.baggs = []
self.areas = []

```

In the `initBoard()` method, we have these lists. The `world` list is having all the objects of the game world. Other lists have only specific objects. For example, the `walls` list contains all the walls of the game.

```

def initWorld(self):

    x = OFFSET
    y = OFFSET
    ...

```

The `initWorld()` method initiates the game world. It goes through the level string and fills the above mentioned lists.

```

elif item == '$':

    b = Baggage(x, y)
    self.baggs.append(b)
    x += SPACE

```

In case of the dollar character, we create a `Baggage` object. The object is appended to the `baggs` list. The `x` variable is increased accordingly.

```

def paintEvent(self, event):
    super().paintEvent(event)

    painter = QPainter()
    painter.begin(self)

```

```

        self.buildWorld(painter)
    painter.end()

```

The `paintEvent()` method is called when the window needs repainting. We delegate the painting to the `buildWorld()` method.

```

def buildWorld(self, painter):
    ...

```

The `buildWorld()` method draws the game world on the window.

```

world = self.walls + self.areas + \
        self.baggs + [self.soko]

```

We create a world list which includes all objects of the game.

```

for i in range(len(world)):

    item = world[i]

    if isinstance(item, Player) or \
        isinstance(item, Baggage):
        painter.drawImage(item.x()+2, item.y()+2,
                           item.getImage())
    else:
        painter.drawImage(item.x(), item.y(),
                           item.getImage())

```

We iterate through the world list and draw the objects. The player and the baggage images are a bit smaller. We add 2 px to their coordinates to center them.

```

if self.completed:
    painter.drawText(25, 20, "Completed")

```

If the level is completed, we draw “Completed” in the upper left corner of the window.

```

if key == Qt.Key_Left:

    if self.checkWallCollision(self.soko,
                                LEFT_COLLISION):
        return

    if self.checkBagCollision(LEFT_COLLISION):
        return

    self.soko.move(-SPACE, 0)

```

Inside the `keyPressEvent()`, we check what keys were pressed. We control the sokoban object with the cursor keys. If we press the left cursor key, we check if the sokoban collides with a wall or with a baggage. If it does not, we move the sokoban to the left.

```

elif key == Qt.Key_R:
    self.restartLevel()

```

The level is restarted if we press the R key.

```

def checkWallCollision(self, actor, _type):

    if _type == LEFT_COLLISION:

```

```

        for i in range(len(self.walls)):
            wall = self.walls[i]
            if actor.isLeftCollision(wall):
                return True
        return False
    ...

```

The `checkWallCollision()` was created to ensure that the sokoban or a baggage do not pass the wall. There are four types of collisions. The above lines check for the left collision.

```

def checkBagCollision(self, _type):

    if _type == LEFT_COLLISION:

        for i in range(len(self.baggs)):
            bag = self.baggs[i]
            if self.soko.isLeftCollision(bag):
                for i in range(len(self.baggs)):
                    item = self.baggs[i]
                    if not bag is item:
                        if bag.isLeftCollision(item):
                            return True
                        if self.checkWallCollision(bag,
                                                    LEFT_COLLISION):
                            return True
                    bag.move(-SPACE, 0)
                    self.isCompleted()
        return False
    ...

```

The `checkBagCollision()` is a bit more involved. A baggage can collide with a wall, with a sokoban object or with another baggage. The baggage can be moved only if it collides with a sokoban and does not collide with another baggage or a wall. When the baggage is moved, it is time to check if the level is completed by calling the `isCompleted()` method.

```

def isCompleted(self):

    num = len(self.baggs)
    completed = 0

    for i in range(num):
        bag = self.baggs[i]
        for j in range(num):
            area = self.areas[j]
            if bag.x() == area.x() and \
               bag.y() == area.y():
                completed += 1

    if completed == num:
        self.completed = True
        self.repaint()

```

The `isCompleted()` method checks if the level is completed. We get the number of bags. We compare the x and y coordinates of all the bags and the destination areas. The game is finished when the `completed` variable equals the number of bags in the game, i.e. we have put all bags on the designated places.

```

def restartLevel(self):

```

```

del self.areas[:]
del self.baggs[:]
del self.walls[:]
self.initWorld()

if self.completed:
    self.completed = False

```

If we do some bad move, we can restart the level. We delete all objects from the important lists and initiate the world again. The completed variable is set to False.

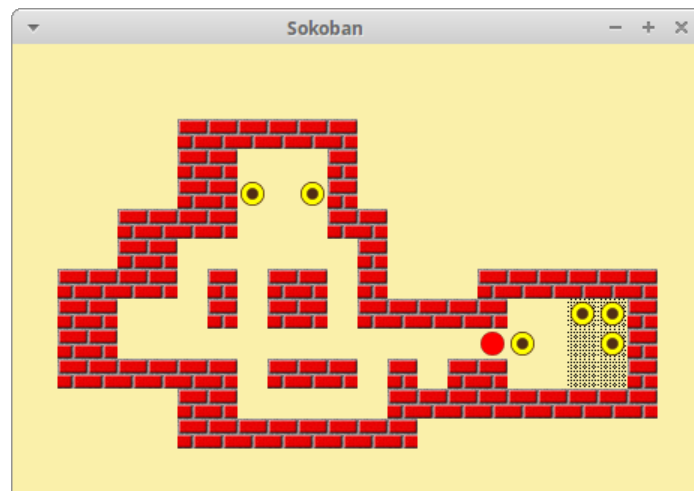


Figure 9.2: Sokoban game

## 9.3 Minesweeper

Minesweeper is a popular board game shipped with many operating systems by default. The goal of the game is to sweep all mines from a minefield. If the player clicks on the cell which contains a mine, the mine detonates and the game is over. Further a cell can contain a number or it can be blank. The number indicates how many mines are adjacent to this particular cell. We set a mark on a cell by right clicking on it. This way we indicate that we believe there is a mine.

Listing 9.3: Minesweeper game

```

#!/usr/bin/python3
# -*- coding: utf-8 -*-

'''
ZetCode Advanced PyQt5 tutorial

This is a simple Mines game clone.

Author: Jan Bodnar

```

*Website: zetcode.com*  
*Last edited: September 2017*  
,,

```
from PyQt5.QtWidgets import (QWidget, QMainWindow,  
    QApplication, QSizePolicy)  
from PyQt5.QtCore import Qt  
from PyQt5.QtGui import QImage, QPainter  
import sys, random
```

```
WIDTH = 250  
HEIGHT = 290
```

```
NUM_IMAGES = 13  
CELL_SIZE = 15
```

```
COVER_FOR_CELL = 10  
MARK_FOR_CELL = 10  
EMPTY_CELL = 0  
MINE_CELL = 9  
COV_MINE_CELL = MINE_CELL + COVER_FOR_CELL;  
MARKED_MINE_CELL = COV_MINE_CELL + MARK_FOR_CELL
```

```
DRAW_MINE = 9  
DRAW_COVER = 10  
DRAW_MARK = 11  
DRAW_WRONG_MARK = 12
```

```
img = [0] * NUM_IMAGES
```

```
class Board(QWidget):
```

```
    def __init__(self, statusbar):  
        super().__init__()
```

```
        self.statusbar = statusbar
```

```
        self.initGame()  
        self.newGame()
```

```
    def initGame(self):
```

```
        self.field = []  
        self.inGame = False  
        self.mines_left = 0
```

```
        self.mines = 40  
        self.rows = 16  
        self.cols = 16  
        self.all_cells = 0
```

```
        for i in range(NUM_IMAGES):  
            img[i] = QImage(str(i) + ".png")
```

```
    def newGame(self):
```

```
        i = 0  
        position = 0  
        cell = 0
```

```

self.inGame = True
self.mines_left = self.mines

self.all_cells = self.rows * self.cols
self.field = [0] * self.all_cells

for i in range(self.all_cells):
    self.field[i] = COVER_FOR_CELL

self.statusbar.showMessage(str(self.mines_left))

i = 0

while i < self.mines:

    position = random.randint(0, self.all_cells)

    if position < self.all_cells and \
        self.field[position] != COV_MINE_CELL:

        current_col = position % self.cols
        self.field[position] = COV_MINE_CELL
        i = i + 1

    if current_col > 0:
        cell = position - 1 - self.cols
        if cell >= 0:
            if self.field[cell] != COV_MINE_CELL:
                self.field[cell] += 1
        cell = position - 1
        if cell >= 0:
            if self.field[cell] != COV_MINE_CELL:
                self.field[cell] += 1

        cell = position + self.cols - 1
        if cell < self.all_cells:
            if self.field[cell] != COV_MINE_CELL:
                self.field[cell] += 1

    cell = position - self.cols
    if cell >= 0:
        if self.field[cell] != COV_MINE_CELL:
            self.field[cell] += 1
    cell = position + self.cols
    if cell < self.all_cells:
        if self.field[cell] != COV_MINE_CELL:
            self.field[cell] += 1

    if current_col < self.cols - 1:
        cell = position - self.cols + 1
        if cell >= 0:
            if self.field[cell] != COV_MINE_CELL:
                self.field[cell] += 1
        cell = position + self.cols + 1
        if cell < self.all_cells:
            if self.field[cell] != COV_MINE_CELL:
                self.field[cell] += 1
        cell = position + 1
        if cell < self.all_cells:
            if self.field[cell] != COV_MINE_CELL:

```

```

        self.field[cell] += 1

def find_empty_cells(self, j):

    current_col = j % self.cols
    cell = 0

    if current_col > 0:
        cell = j - self.cols - 1
        if cell >= 0:
            if self.field[cell] > MINE_CELL:
                self.field[cell] -= COVER_FOR_CELL
            if self.field[cell] == EMPTY_CELL:
                self.find_empty_cells(cell)

        cell = j - 1
        if cell >= 0:
            if self.field[cell] > MINE_CELL:
                self.field[cell] -= COVER_FOR_CELL
            if self.field[cell] == EMPTY_CELL:
                self.find_empty_cells(cell)

        cell = j + self.cols - 1
        if cell < self.all_cells:
            if self.field[cell] > MINE_CELL:
                self.field[cell] -= COVER_FOR_CELL
            if self.field[cell] == EMPTY_CELL:
                self.find_empty_cells(cell)

    cell = j - self.cols
    if cell >= 0:
        if self.field[cell] > MINE_CELL:
            self.field[cell] -= COVER_FOR_CELL
        if self.field[cell] == EMPTY_CELL:
            self.find_empty_cells(cell)

    cell = j + self.cols
    if cell < self.all_cells:
        if self.field[cell] > MINE_CELL:
            self.field[cell] -= COVER_FOR_CELL;
        if self.field[cell] == EMPTY_CELL:
            self.find_empty_cells(cell)

    if current_col < self.cols - 1:
        cell = j - self.cols + 1
        if cell >= 0:
            if self.field[cell] > MINE_CELL:
                self.field[cell] -= COVER_FOR_CELL
            if self.field[cell] == EMPTY_CELL:
                self.find_empty_cells(cell)

        cell = j + self.cols + 1
        if cell < self.all_cells:
            if self.field[cell] > MINE_CELL:
                self.field[cell] -= COVER_FOR_CELL
            if self.field[cell] == EMPTY_CELL:

```



```

        self.find_empty_cells(cell)

    cell = j + 1
    if cell < self.all_cells:
        if self.field[cell] > MINE_CELL:
            self.field[cell] -= COVER_FOR_CELL
        if self.field[cell] == EMPTY_CELL:
            self.find_empty_cells(cell)

def paintEvent(self, event):
    super().paintEvent(event)

    painter = QPainter()
    painter.begin(self)

    self.drawBoard(painter)

    painter.end()

def drawBoard(self, painter):
    cell = 0
    uncover = 0

    for i in range(self.rows):
        for j in range(self.cols):

            cell = self.field[(i * self.cols) + j]

            if not self.inGame:
                if cell == COV_MINE_CELL:
                    cell = DRAW_MINE
                elif cell == MARKED_MINE_CELL:
                    cell = DRAW_MARK
                elif cell > COV_MINE_CELL:
                    cell = DRAW_WRONG_MARK
                elif cell > MINE_CELL:
                    cell = DRAW_COVER

            else:
                if cell > COV_MINE_CELL:
                    cell = DRAW_MARK
                elif cell > MINE_CELL:
                    cell = DRAW_COVER

            painter.drawImage(j*CELL_SIZE, i*CELL_SIZE,
                              img[cell])

def mousePressEvent(self, e):
    x = e.x()
    y = e.y()

    cCol = x // CELL_SIZE
    cRow = y // CELL_SIZE

    rep = False

    if not self.inGame:

```

```

        self.newGame()
        self.repaint()
        return

    pos = (cRow * self.cols) + cCol

    if x < self.cols * CELL_SIZE and \
        y < self.rows * CELL_SIZE:

        if e.button() == Qt.RightButton:

            if self.field[pos] > MINE_CELL:
                rep = True

                if self.field[pos] <= COV_MINE_CELL:
                    if self.mines_left > 0:
                        self.field[pos] += MARK_FOR_CELL
                        self.mines_left = self.mines_left - 1
                        msg = str(self.mines_left)
                        self.statusbar.showMessage(msg)
                    else:
                        msg = "No marks left"
                        self.statusbar.showMessage(msg)
                else:
                    self.field[pos] -= MARK_FOR_CELL;
                    self.mines_left = self.mines_left + 1
                    msg = str(self.mines_left)
                    self.statusbar.showMessage(msg)

            else:

                if self.field[pos] == MARKED_MINE_CELL:
                    return

                if self.field[pos] > MINE_CELL and \
                    self.field[pos] < MARKED_MINE_CELL:

                    self.field[pos] -= COVER_FOR_CELL
                    rep = True

                    if self.field[pos] == MINE_CELL:
                        self.inGame = False
                        self.statusbar.showMessage("Game lost")
                    if self.field[pos] == EMPTY_CELL:
                        self.find_empty_cells(pos)

        uncovered = 0

        for cell in range(self.all_cells):
            if self.field[cell] < MINE_CELL:
                uncovered = uncovered + 1

            if self.mines_left == 0 and uncovered == \
                (self.all_cells - self.mines):
                self.inGame = False
                self.statusbar.showMessage("Game won")

        if rep:
            self.repaint()

```

```

class Mines(QMainWindow):

    def __init__(self):
        super().__init__()

        self.initUI()

    def initUI(self):

        self.setWindowTitle("Minesweeper")

        self.resize(WIDTH, HEIGHT)
        policy = QSizePolicy.Fixed
        self.setSizePolicy(policy, policy)

        self.statusbar = self.statusBar()
        self.setCentralWidget(Board(self.statusbar))

        self.move(300, 300)
        self.show()

def main():

    app = QApplication([])
    mines = Mines()
    sys.exit(app.exec_())

if __name__ == '__main__':
    main()

```

---

This is the code for the Minesweeper game in PyQt5.

```

NUM_IMAGES = 13
CELL_SIZE = 15

```

There are 13 images used in this game. A cell can be surrounded by maximum of 8 mines, so we need numbers 1..8. We need images for an empty cell, a mine, a covered cell, a marked cell and finally for a wrongly marked cell. The size of each of the images is 15x15 px.

```

COVER_FOR_CELL = 10
MARK_FOR_CELL = 10
EMPTY_CELL = 0
...

```

A mine field is a list of numbers. For example 0 denotes an empty cell. Number 10 is used for a cell cover as well as for a mark. Using constants improves readability of the code.

```

DRAW_MINE = 9
DRAW_COVER = 10
DRAW_MARK = 11
DRAW_WRONG_MARK = 12

```

These constants are used inside the `drawBoard()` method to draw the cells of a minefield. They are indexes into the `img` image list.

```

self.field = []

```

The `field` is a list of numbers. Each cell in the field has a specific number. A mine cell has number 9. A cell with number 2, meaning it is adjacent to two mines, has number two. The numbers are added. For example, a covered mine has number 19, 9 for the mine and 10 for the cell cover.

```
self.mines = 40
self.rows = 16
self.cols = 16
```

The minefield in our game has 40 hidden mines. There are 16 rows and 16 columns in this field, so there are 256 cells together in the minefield.

```
for i in range(NUM_IMAGES):
    img[i] = QImage(str(i) + ".png")
```

Here we load our images into the `img` list. The images are named 0.png, 1.png ... 12.png.

```
self.all_cells = self.rows * self.cols
self.field = [0] * self.all_cells
```

```
for i in range(self.all_cells):
    self.field[i] = COVER_FOR_CELL
```

In the `newGame()` method we start the game. We set up the minefield. The for loop creates a cover for each cell.

```
while i < self.mines:

    position = random.randint(0, self.all_cells)

    if position < self.all_cells and \
        self.field[position] != COV_MINE_CELL:

        current_col = position % self.cols
        self.field[position] = COV_MINE_CELL
```

In the while cycle we randomly position all mines in the field.

```
if current_col > 0:
    cell = position - 1 - self.cols
    if cell >= 0:
        if self.field[cell] != COV_MINE_CELL:
            self.field[cell] += 1
```

After we have added a new mine to the field, we must create numbers around the mine. Each of the cells can be surrounded by 8 cells maximum. (This does not apply to the border cells.) We raise the number for adjacent cells for the randomly placed mine. In our example, we add 1 to the top neighbor of the cell in question.

```
cell = j - 1
if cell >= 0:
    if self.field[cell] > MINE_CELL:
        self.field[cell] -= COVER_FOR_CELL
    if self.field[cell] == EMPTY_CELL:
        self.find_empty_cells(cell)
```

In the `find_empty_cells()` method we find empty cells. If the player clicks on a mine cell, the game is over. If he clicks on a cell adjacent to a mine, he uncovers a number indicating how many mines the cell is adjacent to. Clicking on an

empty cell leads to uncovering many other empty cells plus cells with a number that form a border around empty cells. We use a recursive algorithm to find empty cells. In the above code, we check the cell that is located to the left of an empty cell in question. If it is not empty, it is uncovered. If it is empty, the process is repeated by recursively calling the `find_empty_cells()` method.

```
...
else:
    if cell > COV_MINE_CELL:
        cell = DRAW_MARK
    elif cell > MINE_CELL:
        cell = DRAW_COVER
...
painter.drawImage(j*CELL_SIZE, i*CELL_SIZE,
                  img[cell])
```

The `drawBoard()` method turns numbers into images. In the above code, we check what number is in the current cell of the field. Depending on this number, we draw an appropriate image for the cell.

```
def mousePressEvent(self, e):
    ...
```

In the `mousePressEvent()` method we react to mouse clicks. The Minesweeper game is controlled solely by mouse: we react to left and right mouse clicks.

```
x = e.x()
y = e.y()

cCol = x // CELL_SIZE
cRow = y // CELL_SIZE
```

First, we retrieve the x and y coordinates of a mouse click. If we divide these two numbers by the cell size, we get the current column and current row.

```
rep = False
```

The `rep` variable controls the repainting of the board. The Minesweeper game is mostly static. For example, we do not need to repaint the board if we have clicked on the empty cell or we have left clicked on a marked cell.

```
pos = (cRow * self.cols) + cCol
```

Using the current row and the current column, we calculate a position into the field. This position is an index used to get the cell on which we have clicked.

```
if x < self.cols * CELL_SIZE and \
    y < self.rows * CELL_SIZE:
```

We continue only if we have clicked inside the minefield.

```
if self.field[pos] <= COV_MINE_CELL:
    if self.mines_left > 0:
        self.field[pos] += MARK_FOR_CELL
        self.mines_left = self.mines_left - 1
        msg = str(self.mines_left)
        self.statusbar.showMessage(msg)
    else:
        msg = "No marks left"
        self.statusbar.showMessage(msg)
```

These lines determine what happens if we right click on a cell that is not already marked by a user to be a mine cell. If there are still some mines left, we add a `MARK_FOR_CELL` value to a current cell, decrease the `mines_left` and update the statusbar. Otherwise, we only show a “No marks left” message in the statusbar.

```
else:
    self.field[pos] -= MARK_FOR_CELL;
    self.mines_left = self.mines_left+1
    msg = str(self.mines_left)
    self.statusbar.showMessage(msg)
```

We can change our mind as for whether a cell is supposed to be a mine cell. By right clicking on a marked cell, we remove the mark and increase the `mines_left` variable.

```
if self.field[pos] == MARKED_MINE_CELL:
    return
```

Nothing happens if we left click on the marked cell. If we want to uncover a marked cell, we first have to remove the mark with another right click.

```
self.field[pos] -= COVER_FOR_CELL
```

Left click removes a cover from the cell.

```
if self.field[pos] == MINE_CELL:
    self.inGame = False
    self.statusbar.showMessage("Game lost")
```

If we uncover a mine, the game is over.

```
if self.field[pos] == EMPTY_CELL:
    self.find_empty_cells(pos)
```

If we click on an empty cell, we call the `find_empty_cells()` method which recursively looks for all adjacent empty cells.

```
uncovered = 0
for cell in range(self.all_cells):
    if self.field[cell] < MINE_CELL:
        uncovered = uncovered + 1

    if self.mines_left == 0 and uncovered == \
        (self.all_cells - self.mines):
        self.inGame = False
        self.statusbar.showMessage("Game won")
```

These lines check for successful game over. Cells with numbers lower than `MINE_CELL` are either empty cells or cells which are adjacent to a mine, having numbers 1..8. All these cells, without a cover, form a list of uncovered cells. The game is won if there are no more (marked) mines left and the number of uncovered cells is equal to all cells minus mine cells.



Figure 9.3: Minesweeper game

## Chapter 10

# References

1. [Qt Online Reference Documentation](#)
2. [Free Linux applications](#)
3. [RFC Documents](#)



# Index

- addItem(), 67
- addLayout(), 95
- addressEntries(), 188
- addresses(), 190
- addStretch(), 97
- addText(), 46
- addToGroup(), 67
- addWidget(), 90, 108, 110, 112
- allInterfaces(), 187
- appendRow(), 143
- authentication, 196–197
- authenticationRequired, 196, 197
  
- bindValue(), 169
- Blocking and non-blocking requests, 199
- boundingRect(), 66
  
- center(), 76
- childItems(), 67
- collidingItems(), 54, 57, 86
- columnCount(), 144
- commit(), 174, 176
- comparing dates, 4
- connecting to an SMTP server, 212–214
- contains(), 31, 34
- count(), 173
- createEditor(), 156, 158
- createPolygon(), 56
- current date and time, 2–3
- currentDateTime(), 3
- custom widgets, 116–130
  - CPU widget, 120–123
  - led widget, 116–120
  - thermometer widget, 124–130
  
- data(), 149
- dataChanged, 152
- date and time, 1–11
- datetime arithmetic, 6–7
- day of week, month, year, 5–6
- day(), 5
  
- daylight saving time, 7–8
- dayOfWeek(), 4, 5
- dayOfYear(), 5
- daysInMonth(), 5
- daysInYear(), 5
- daysTo(), 6
- delegate, 155
- destroyItemGroup(), 67
- difference in days, 6
- drawEllipse(), 22, 41
- drawLine(), 12, 14
- drawPie(), 71
- drawRect(), 161
- drawText(), 17
  
- echo service, 204–208
- error(), 189
- errorString(), 192
- exec\_(), 165
  
- fetching a favicon, 198–199
- fieldName(), 173
- fillRect(), 23
- filterItems(), 155
- first(), 166
- flags(), 149
- fromSecsSinceEpoch(), 9
  
- games, 218–248
- get(), 192, 199
- graphics, 12–44
  - clipping, 38–41
  - donut example, 19–21
  - drawing text, 15
  - gradients, 24–26
  - grayscale image, 17
  - hit test, 31–34
  - image reflection, 34–38
  - lines, 12–15
  - moving star, 41–44
  - puff effect, 29–31
  - shapes, 21–23
  - transparent rectangles, 23–24

- waiting effect, 26–29
- graphics view framework, 45–87
  - aliens, 78–87
  - collision detection, 54–58
  - custom graphics item, 47–49
  - grouping items, 63–68
  - item animation, 52
  - progress meter, 68–72
  - rotating arrow, 72–78
  - rotation, 49–52
  - selecting items, 58–61
  - simple example, 45–47
  - zooming items, 61–63
- hardwareAddress(), 187
- hasFeature(), 171
- head(), 195
- hostName, 189
- indexes(), 146
- initializing date and time, 1–2
- initUI(), 179
- IP address, host name, 188–191
- isDaylightTime(), 8
- isLeapYear(), 8
- items(), 85
- Julian day, 10–11
- keyPressEvent(), 224
- lastError(), 163, 165, 166
- layout management, 88–115
  - absolute positioning, 88–90
  - alignment, 98–101
  - box layout, 90–91
  - buttons example, 101–102
  - findreplace example, 104
  - minimum and maximum size, 93–95
  - nesting layout managers, 94
  - QGridLayout, 107
  - size hint, 92
  - size policy, 91–93
  - stretch factor, 96–98
  - windows example, 102–104
- leap year, 8–9
- lookupHost(), 189
- mapToScene(), 76
- metadata, 170–174
- Minesweeper game, 238–248
- model and view, 131–162, 176–185
  - basic examples, 131–136
  - QListView, 132–134
  - QTableView, 134–136
  - QTreeView, 131–132
  - delegates, 155–162
  - filtering data, 152–155
  - QItemSelectionModel, 144–147
  - QModelIndex, 141–144
  - sorting, 136–141
    - built-in sorting, 136–139
    - proxy models, 139–141
    - subclassing models, 147–152
- mouseMoveEvent(), 76
- mousePressEvent(), 34
- network interfaces, 186–188
- networking, 186–217
- next(), 167, 168
- number of days, 5
- numRowsAffected(), 173, 174
- offsetFromUtc(), 3
- open(), 165
- paint(), 57, 71, 161
- paintEvent(), 16, 119, 127, 223, 236
- pixel(), 19
- prepare(), 169, 170
- pyqtProperty, 54
- QAbstractItemModel, 147
- QAbstractListModel, 147, 149
- QAbstractTableModel, 147
- QAuthenticator, 197
- QBasicTimer, 78, 85
- QByteArray, 194
- QCheckBox, 106, 138
- QComboBox, 105
- QDate, 1
- QDateTime, 1, 9
- QFileSystemModel, 132
- QFontMetrics, 31
- QFrame, 110
- QGraphicsItem, 45, 54, 56, 70
- QGraphicsItemGroup, 63, 66
- QGraphicsPixmapItem, 76
- QGraphicsRectItem, 48
- QGraphicsScene, 45, 46, 54
- QGraphicsTextItem, 84, 85
- QGraphicsView, 45, 46, 51
- qGray(), 19
- QGridLayout, 88
- QGroupBox, 106
- QHBoxLayout, 88, 90, 91
- QHostInfo, 188
- QHVBoxLayout, 88
- QImage, 37

- QItemSelectionModel, 144
- QLabel, 88, 105, 115
- QLinearGradient, 24, 128, 129
- QLineEdit, 111, 113
- QListView, 113–115, 131–134, 140, 149, 150, 154
- QMainWindow, 178
- QModelIndex, 143
- QNetworkAccessManager, 199
- QNetworkAccessManager, 186, 191–192
- QNetworkInterface, 186
- QNetworkReply, 191, 192
- QNetworkRequest, 191, 192
- QObject, 52, 54
- QPainter, 12, 45
- QPainterPath, 22, 43
- QPoint, 76
- QPropertyAnimation, 52, 54
- QPushButton, 93, 101, 106, 115, 138
- QRadialGradient, 25, 71, 129
- QRect, 31
- QRegExp, 155
- QSizePolicy, 91
- QSizePolicy.Fixed, 91, 103
- QSlider, 122, 127
- QSortFilterProxyModel, 140, 152, 154
- QSpinBox, 158
- QSqlDatabase, 174
- QSqlQuery, 164–170, 173
- QSqlQueryModel, 177–179
- QSqlRecord, 173
- QSqlRelationalTableModel, 177, 182, 185
- QSqlTableModel, 177, 180, 182
- QStandardItem, 136, 143
- QStandardItemModel, 136, 141
- QStandartItemModel, 143
- QStringListModel, 132, 134, 140
- QStyledItemDelegate, 156, 157
- QSvgRenderer, 119
- Qt.AlignBottom, 100
- Qt.AlignLeft, 99
- Qt.ItemIsEditable, 152
- Qt.ItemIsEnabled, 150
- Qt.ItemIsSelectable, 150
- QTableView, 131, 134, 136, 138, 146, 178–180, 182
- QTcpServer, 186, 208
- QTcpSocket, 186
- QTextEdit, 94
- QThread, 33
- QTime, 1
- Qtnetwork, 186, 199
- QTreeView, 131, 132, 141, 143
- QtSql, 165
- QtSql module, 163–185
- quadTo(), 128
- QUdpSocket, 186
- QVariant, 150
- QVBoxLayout, 90, 106
- QWidget, 118, 122
- QXmlStreamReader, 216
- rawHeaderPairs, 195
- readAll(), 192
- removeItem(), 60
- resetTransform(), 63
- rollback(), 174, 176
- rotate(), 51
- row(), 143
- rowCount(), 149, 150
- run(), 34
- scale(), 63, 127
- setChanged(), 60
- select(), 185
- selectedItems(), 60, 67
- selection(), 146
- selectionModel(), 146
- sender(), 120
- setAlignment(), 98–100
- setClipRect(), 40
- setData(), 152
- setDuration(), 54
- setDynamicSortFilter(), 154
- setEditorData(), 156, 158
- setEndValue(), 54
- setFilterRegExp(), 155
- setFont(), 16
- setGeometry(), 89
- setHorizontalHeaderLabels(), 143
- setHorizontalSpacing(), 108, 110
- setItem(), 147
- setKeyValueAt(), 54
- setModel(), 132, 134
- setModelData(), 156
- setMouseTracking(), 76
- setOpacity(), 23
- setPixel(), 19
- setPlainText(), 85
- setPos(), 83
- setRelation(), 185
- setRootIsDecorated(), 143
- setRootPath(), 132
- setScene(), 46
- setSceneRect(), 51
- setSizePolicy(), 91

- setSocketDescriptor(), 206
- setSortingEnabled(), 138
- setSourceModel, 140
- setStartValue(), 54
- setTable(), 185
- setVerticalSpacing(), 108, 110
- sizeHint(), 92, 93, 160
- Snake game, 218–225
- Sokoban game, 225–238
- sort(), 141
- SQLite database, 163–164
- start(), 72
- text(), 120
- timerEvent(), 31, 85
- timeZoneAbbreviation(), 8
- toJulianDay(), 10
- toLocalTime(), 3
- toSecsSinceEpoch(), 9
- toString(), 2
- toUTC(), 3
- transaction(), 176
- transactions, 174–176
- universal time, 3
- Unix epoch, 9–10
- updateEditorGeometry(), 156, 158
- value(), 166, 168
- waitForReadyRead(), 201
- weather conditions, 214–217
- web server, 208–212
- weekday, 4–5
- write(), 201, 206