1. GUI, PyQt5 ბიბლიოთეკა, ეტიკეტები, ღილაკები, Qt Designer, ComboBox

პროგრამების უმეტესობა არის ტექსტზე დაფუძნებული, მაგრამ ბევრ აპლიკაციას სჭირდება GUI (გრაფიკული მომხმარებლის ინტერფეისი).

პითონი გვთავაზობს რამდენიმე სხვადასხვა ვარიანტს GUI-ზე დაფუძნებული პროგრამების დასაწერად. მაგ:

Tkinter: ერთერთი ყველაზე მარტივად შესასწავლი. Tkinter არის პითონის სტანდარტული GUI (გრაფიკული მომხმარებლის ინტერფეისი). ეს არის ყველაზე ხშირად გამოყენებული ინსტრუმენტარიუმი პითონში GUI პროგრამირებისთვის.

JPython: ეს არის პითონის პლატფორმა Java-სთვის, რომელიც უზრუნველყოფს Python სკრიპტების უწყვეტ წვდომას Java კლასის ბიბლიოთეკებში ადგილობრივი აპარატისთვის.

wxPython: ეს არის ღია წყაროს, მრავალ-პლატფორმული GUI ინსტრუმენტი, რომელიც დაწერილია C++-ში. ეს არის Tkinter-ის ერთ-ერთი ალტერნატივა, რომელიც მუშაობს პითონთან.

PyQt5: არის GUI ვიჯეტების ინსტრუმენტარიუმის უახლესი ვერსია, რომელიც შემუშავებულია Riverbank Computing-ის მიერ. ეს არის Python ინტერფეისი Qt-სთვის, ერთ-ერთი ყველაზე ძლიერი და პოპულარული კროს-პლატფორმული GUI ბიბლიოთეკა. PyQt5 არის პითონის პროგრამირების ენისა და Qt ბიბლიოთეკის ნაზავი.

PyQt არის GUI ვიჯეტების ხელსაწყოების ნაკრები. ეს არის Python-ის ინტერფეისი Qt-სთვის, ერთ-ერთი ყველაზე ძლიერი და პოპულარული მრავალ პლატფორმული GUI ბიბლიოთეკა. PyQt შეიქმნა RiverBank Computing Ltd-ის მიერ. PyQt5 არის უახლესი ვერსია. მისი ჩამოტვირთვა შესაძლებელია მისი ოფიციალური ვებგვერდიდან − ivebankcomputing.com.
PyQt API არის მოდულების ნაკრები, რომელიც შეიცავს კლასებისა და ფუნქციების დიდ რაოდენობას. QtGui მოდული შეიცავს ყველა გრაფიკულ კონტროლს. გარდა ამისა, არსებობს XML (QtXml), SVG (QtSvg) და SQL (QtSql) მუშაობის მოდულები და ა.შ.
ხშირად გამოყენებული მოდულების სია მოცემულია ქვემოთ:

QtCore: ძირითადი არა-GUI კლასები, რომლებიც გამოიყენება სხვა მოდულების მიერ
QtGui: მომხმარებლის გრაფიკული ინტერფეისის კომპონენტები
QtMultimedia: დაბალი დონის მულტიმედიური პროგრამირების კლასები
QtNetwork: ქსელის პროგრამირების კლასები
QtOpenGL: OpenGL მხარდაჭერის კლასები
QtScript: კლასები Qt სკრიპტების შესაფასებლად
QtSql: კლასები მონაცემთა ბაზის ინტეგრაციისთვის SQL-ის გამოყენებით
QtSvg: კლასები SVG ფაილების შიგთავსის ჩვენებისთვის
QtWebKit: კლასები HTML-ის რენდერისა და რედაქტირებისთვის
QtXml: XML დამუშავების კლასები
QtWidgets: კლასები კლასიკური დესკტოპის სტილის ინტერფეისების შესაქმნელად
QtDesigner: კლასები Qt Designer-ის გაფართოებისთვის

PyQt თავსებადია ყველა პოპულარულ ოპერაციულ სისტემასთან, მათ შორის Windows, Linux და Mac OS. არის ორმაგი ლიცენზიით და ხელმისაწვდომია როგორც GPL, ასევე კომერციული ლიცენზიით. უახლესი სტაბილური ვერსიაა PyQt5-5.13.2.

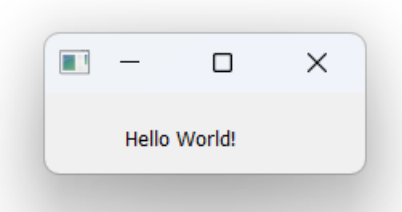
PyQt ინსტალაციის რეკომენდებული გზა არის PIP ინსტრუქციის გამოყენება:
pip3 install PyQt5
Qt Designer, PyQt5 სამუშაო ინსტრუმენტების დასაყენებლად ვიძახებთ:
pip3 install pyqt5-tools

მარტივი GUI აპლიკაციის შექმნა PyQt-ის გამოყენებით მოიცავს შემდეგ ნაბიჯებს:
 QtCore, QtGui და QtWidgets მოდულების შემოყვანა PyQt5 პაკეტიდან.
 QApplication კლასის აპლიკაციის ობიექტის შექმნა.
 QWidget ობიექტი ქმნის მთავარ ფანჯარას. სადაც ემატება QLabel ობიექტი.
 QLabel-ში ემატება ეტიკეტის წარწერა, როგორც "hello world".
 setGeometry() მეთოდით განისაზღვრება ფანჯრის ზომა და პოზიცია .
 app.exec_() მეთოდით იქმნება აპლიკაციის მთავარი ციკლი .

PyQt-ში Hello World პროგრამის კოდი:

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
def window():
 app = QApplication(sys.argv)
 w = QWidget()
 b = QLabel(w)
 b.setText("Hello World!")
 w.setGeometry(100,100,200,50)
 b.move(50,20)
 w.setWindowTitle("PyQt5")
 w.show()
 sys.exit(app.exec_())
if __name__ == '__main__':
 window()
```

შდეგი:

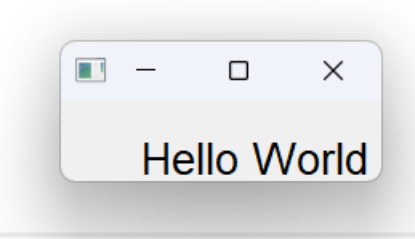


იგივე ამოცანის შესრულება შესაძლებელია ობიექტზე ორიენტირებული მიდგომით.
 QtCore, QtGui და QtWidgets მოდულების შემოყვანა PyQt5 პაკეტიდან.
 QApplication კლასის აპლიკაციის ობიექტის შექმნა.

 window კლასის გამოცხადება QWidget კლასზე დაყრდნობით
 QLabel-ში ახალი ობიექტის დამატება და ლეიბლში "hello world"-ის დამატება.
 ფანჯრის ზომის და პოზიციის განსაზღვრა setGeometry() მეთოდით.
 აპლიკაციის მთავარი ციკლის შემოყვანა app.exec_() მეთოდით.
ქვემოთ მოცემულია ობიექტზე ორიენტირებული გადაწყვეტის სრული კოდი:

```
import sys
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *
class window(QWidget):
    def __init__(self, parent = None):
        super(window, self).__init__(parent)
        self.resize(200,50)
        self.setWindowTitle("PyQt5")
        self.label = QLabel(self)
        self.label.setText("Hello World")
        font = QFont()
        font.setFamily("Arial")
        font.setPointSize(16)
        self.label.setFont(font)
        self.label.move(50,20)
def main():
    app = QApplication(sys.argv)
    ex = window()
    ex.show()
    sys.exit(app.exec_())
if __name__ == '__main__':
 main()
```

გამომავალი:

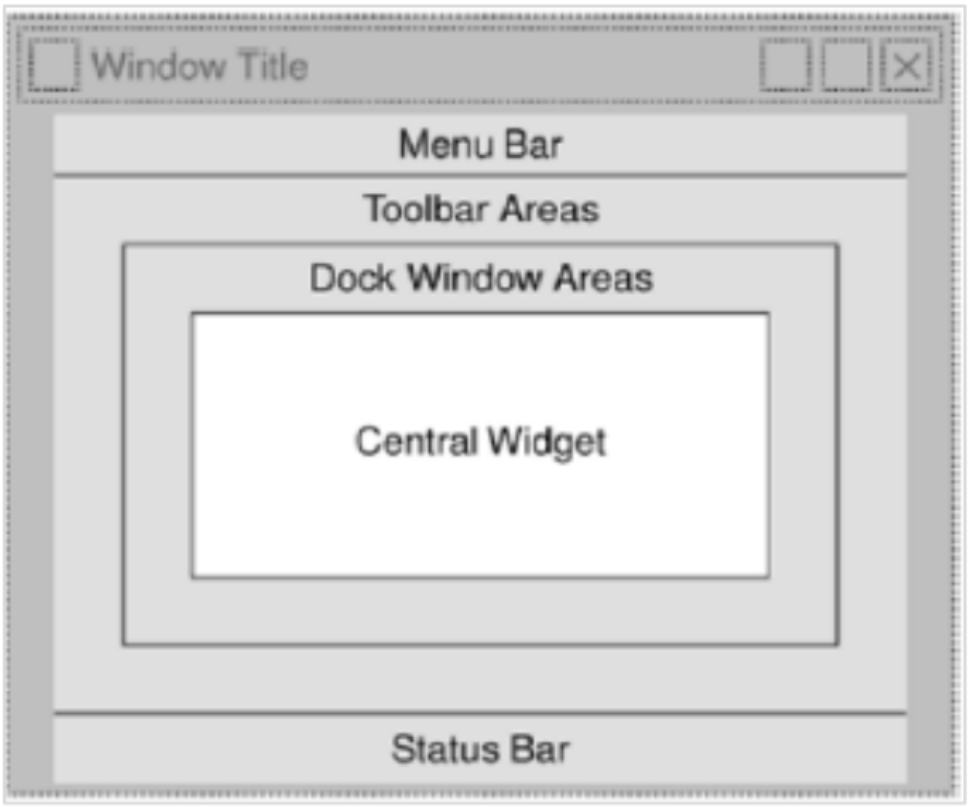


PyQt API შეიცავს 400-ზე მეტ კლასს. QObject კლასი არის კლასის იერარქიის სათავეში. ეს არის ყველა Qt ობიექტის საბაზისო კლასი. გარდა ამისა, QPaintDevice კლასი არის საბაზისო კლასი ყველა ობიექტისთვის, რომელიც შეიძლება დაიხატოს.
QApplication კლასი მართავს GUI აპლიკაციის ძირითად პარამეტრებს და კონტროლს. ის შეიცავს მოვლენების მთავარ ციკლს, რომლის შიგნითაც ხდება ფანჯრის ელემენტებისა და სხვა წყაროების მიერ წარმოქმნილი მოვლენების დამუშავება. ის ასევე ამუშავებს აპლიკაციის პარამეტრებს სისტემის მასშტაბით.

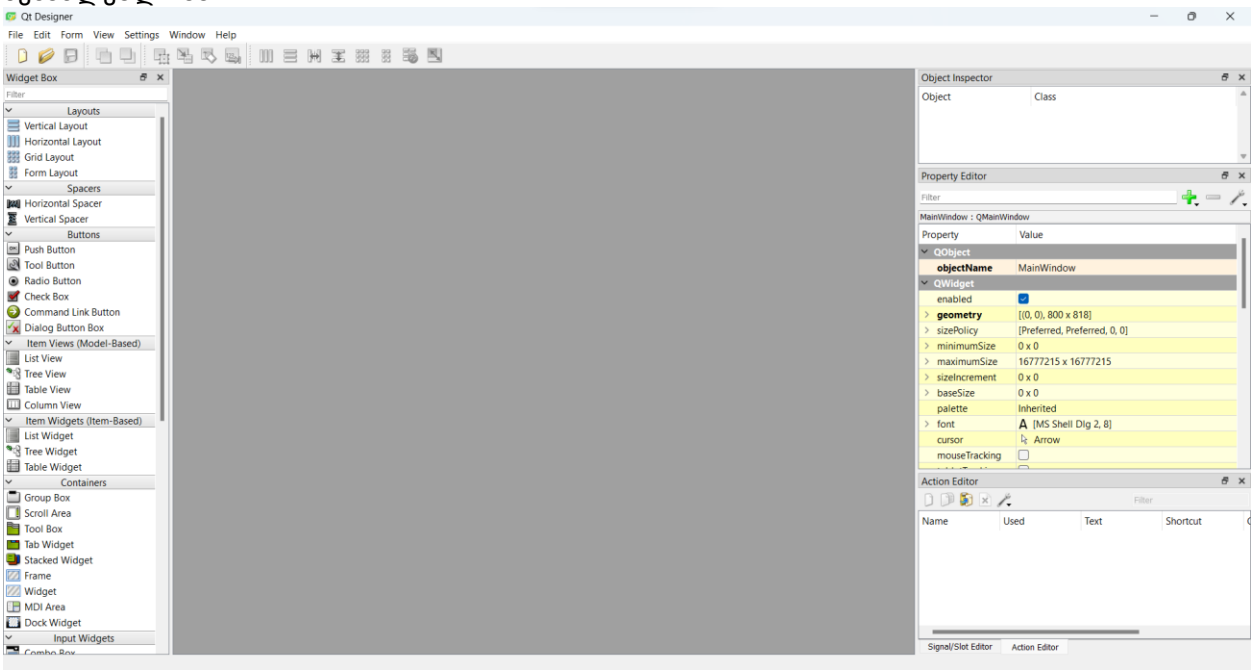
QWidget კლასი, წარმოებულია QObject და QPaintDevice კლასებიდან არის საბაზისო კლასი მომხმარებლის ინტერფეისის ყველა ობიექტისთვის. QDialog და QFrame კლასები ასევე მიღებულია QWidget კლასიდან. მათ აქვთ საკუთარი ქვეკლასების სისტემა.
აქ მოცემულია ხშირად გამოყენებული ვიჯეტების სია:

1 QLabel - გამოიყენება ტექსტის ან სურათის საჩვენებლად
2 QLineEdit - საშუალებას აძლევს მომხმარებელს შეიყვანოს ტექსტის ერთი ხაზი
3 QTextEdit - საშუალებას აძლევს მომხმარებელს შეიყვანოს მრავალსტრიქონიანი ტექსტი
4 QPushButton - ბრძანების ღილაკი მოქმედების გამოსაძახებლად
5 QRadioButton - საშუალებას აძლევს მომხმარებელს რამდენიმე ვარიანტიდან აირჩიოს ერთი
6 QCheckBox - საშუალებას აძლევს მომხმარებელს აირჩიოს ერთზე მეტი ვარიანტი
7 QSpinBox - საშუალებას აძლევს მომხმარებელს გაზარდოს/შეამციროს მთელი რიცხვი
8 QScrollBar - იძლევა ვიჯეტის შიგთავსზე წვდომას ეკრანის დიაფრაგმის მიღმა
9 QSlider - საშუალებას აძლევს მომხმარებელს წრფივად შეცვალოს დაკავშირებული მნიშვნელობა
10 QComboBox - გთავაზობთ ელემენტების ჩამოსაშლელ სიას ასარჩევად
11 QMenuBar - ჰორიზონტალური ზოლი, რომელიც შეიცავს QMenu ობიექტებს
12 QStatus Bar - ჩვეულებრივ QMainWindow-ის ბოლოში, იძლევა სტატუსის ინფორმაცია.
13 QToolBar - ჩვეულებრივ QMainWindow-ის თავზე შეიცავს მოქმედების ღილაკებს
14 QListView - გთავაზობთ ელემენტების არჩევით სიას ListMode-ში ან IconMode-ში
15 QPixmap - გამოიყენება გამოსახულების QLabel ან QPushButton ობიექტზე გამოსატანად
16 QDialog - მოდალური ან არამოდალური ფანჯარა, რომელსაც შეუძლია ინფორმაციის დაბრუნება მშობელ ფანჯარაში

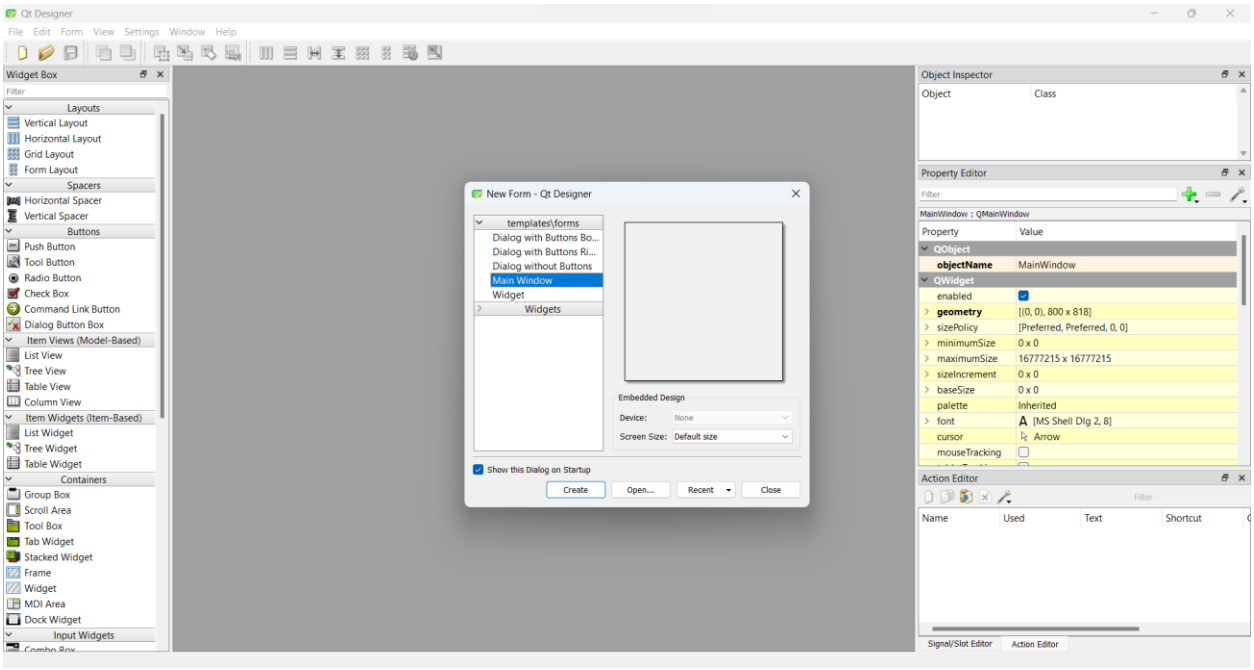
ტიპიური GUI აპლიკაციის უმაღლესი დონის ფანჯარა იქმნება QMainWindow ვიჯეტის ობიექტით. ზოგიერთი ვიჯეტი, როგორც ზემოთ ჩამოთვლილი, იკავებს მისთვის დანიშნულ ადგილს ამ მთავარ ფანჯარაში, ზოგი კი მოთავსებულია ვიჯეტის ცენტრალურ ზონაში სხვადასხვა განლაგების მენეჯერების გამოყენებით.
შემდეგ დიაგრამაზე ნაჩვენებია QMainWindow ჩარჩოს სტრუქტურა:



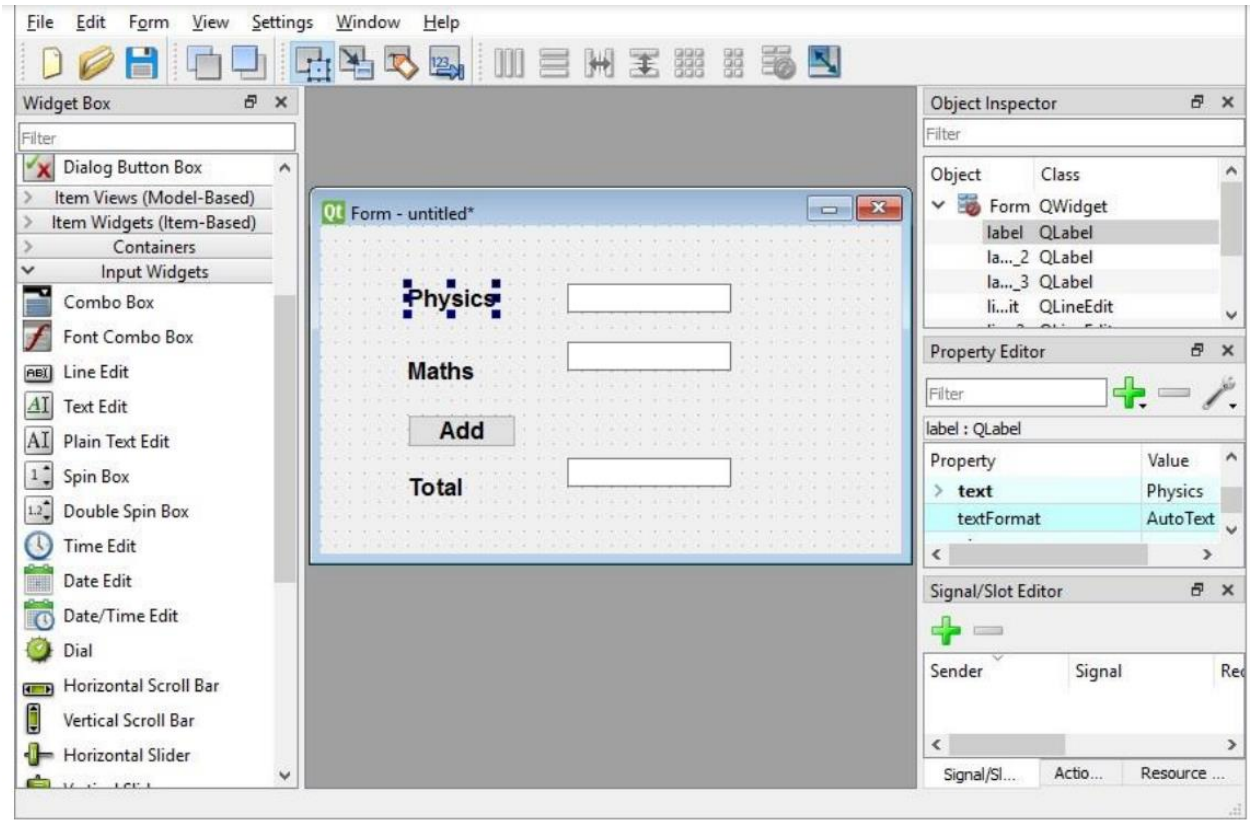
PyQt ინსტალერს გააჩნია GUI შესაქმნელი ინსტრუმენტი, რომელსაც ეწოდება Qt Designer. მისი მარტივი გადაადგილების ინტერფეისის გამოყენებით, GUI ინტერფეისი შეიძლება სწრაფად შეიქმნას კოდის დაწერის გარეშე. თუმცა, ეს არ არის ინტერპრეტატორი, როგორიცაა Visual Studio. აქედან გამომდინარე, Qt დიზაინერს არ აქვს აპლიკაციის გამართვისა და შექმნის შესაძლებლობა.



დაიწყეთ GUI ინტერფეისის დიზაინის შექმნა File -> New მენიუს არჩევით.



შემდეგ შეგიძლიათ გადაათრიოთ და ჩააგდოთ საჭირო ვიჯეტები მარცხენა პანელზე განთავსებული ვიჯეტების ველიდან. თქვენ ასევე შეგიძლიათ მიანიჭოთ მნიშვნელობა ფორმაზე განთავსებული ვიჯეტის თვისებებს.



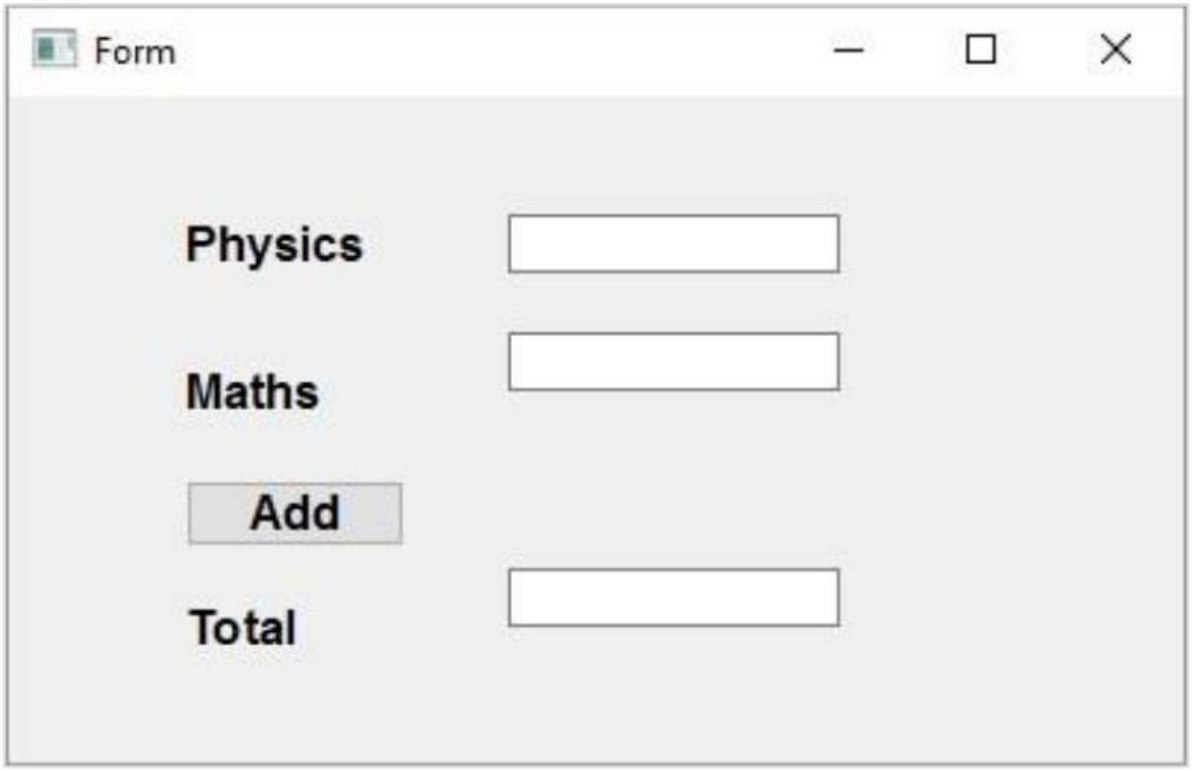
შექმნილი ფორმა შევინახოთ როგორც demo.ui. ეს UI ფაილი შეიცავს ვიჯეტების XML წარმოდგენას და მათ თვისებებს დიზაინში. ეს დიზაინი ითარგმნება პითონის ფაილში, pyuic5 ბრძანების ხაზის გამოყენებით. pyuic5-ის გამოყენება შემდეგია:

pyuic5 -x demo.ui -o demo.py

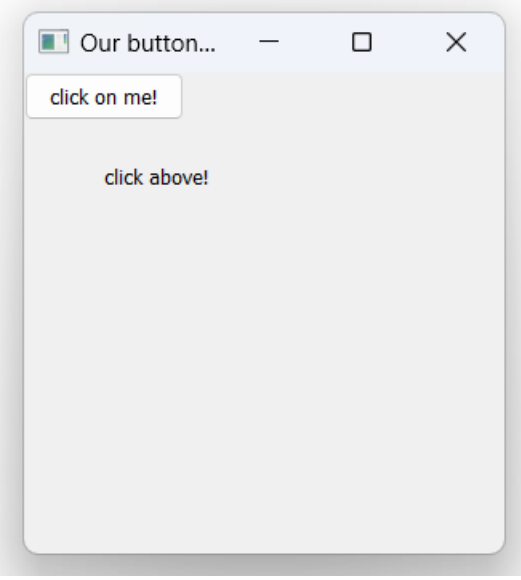
ზემოთ მოყვანილ ბრძანებაში, -x გადამრთველი ამატებს მცირე რაოდენობის კოდს გენერირებულ Python სკრიპტში (XML-დან) ისე, რომ ის გახდეს თვითშესრულებადი დამოუკიდებელი აპლიკაცია.

```
if __name__ == "__main__":
    import sys
    app = QtGui.QApplication(sys.argv)
    Dialog = QtGui.QDialog()
    ui = Ui_Dialog()
    ui.setupUi(Dialog)
    Dialog.show()
    sys.exit(app.exec_())
```

შედეგად მიღებული პითონის სკრიპტი შესრულებულია შემდეგი დიალოგური ფანჯრის საჩვენებლად:



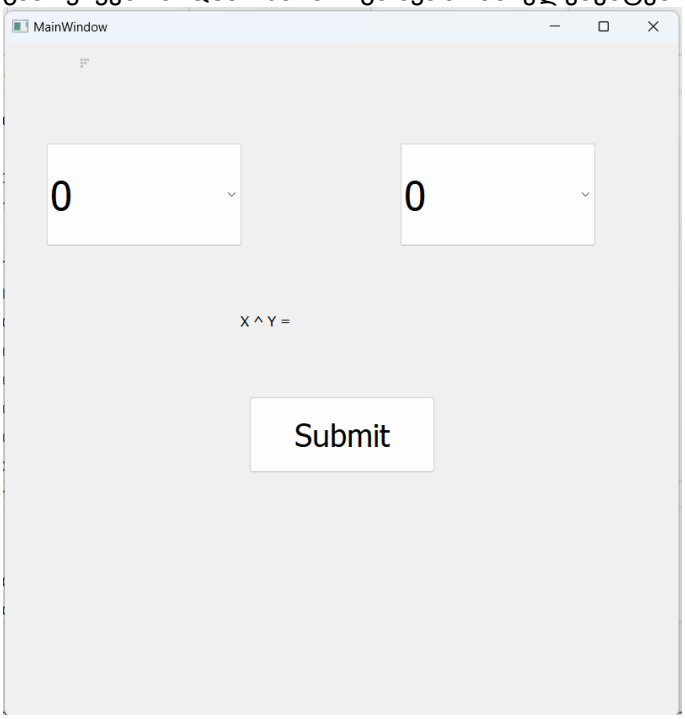
მომხმარებელს შეუძლია შეიყვანოს მონაცემები შეყვანის ველებში, მაგრამ დამატება ღილაკზე დაწკაპუნება არ გამოიწვევს რაიმე მოქმედებას, რადგან ის არ არის დაკავშირებული რაიმე ფუნქციასთან. მომხმარებლის მიერ გენერირებულ მოქმედებაზე რეაგირებას მოვლენის მართვა ეწოდება.
კონსოლის რეჟიმის აპლიკაციისგან განსხვავებით, რომელიც სრულდება თანმიმდევრულად, GUI-ზე დაფუძნებული აპლიკაცია იმართება მოვლენებით. ფუნქციები ან მეთოდები სრულდება მომხმარებლის ქმედებების საპასუხოდ, როგორიცაა ღილაკზე დაწკაპუნება, ელემენტის არჩევა კოლექციიდან ან მაუსის დაწკაპუნება და ა.შ., რასაც ეწოდება მოვლენები.
ვიჯეტები, რომლებიც გამოიყენება GUI ინტერფეისის შესაქმნელად, მოქმედებს, როგორც ასეთი მოვლენების წყარო. თითოეული PyQt ვიჯეტი, რომელიც მიღებულია QObject კლასიდან, შექმნილია იმისათვის, რომ გამოსცეს "სიგნალი" ერთი ან მეტი მოვლენის საპასუხოდ. სიგნალი თავისთავად არ ასრულებს რაიმე მოქმედებას. ამის ნაცვლად, ის "დაკავშირებულია" ე.წ. "სლოტთან". სლოტი შეიძლება იყოს Python-ის ნებისმიერი გამოსაძახებელი ფუნქცია.
`click on me!` ღილაკი თავისთავად ვერ შეასრულებს რაიმე მოქმედებას



მოცემული კოდი აკავშირებს `b1` ღილაკს ჩვენს მიერ შექმნილ `button_clicked` მეთოდთან
`self.b1.clicked.connect(self.button_clicked)`

ღილაკზე დაწკაპუნების შედეგად გაიშვება `button_clicked` მეთოდი

გამოვიყენოთ QComboBox - გთავაზობთ ელემენტების ჩამოსაშლელ სიას ასარჩევად



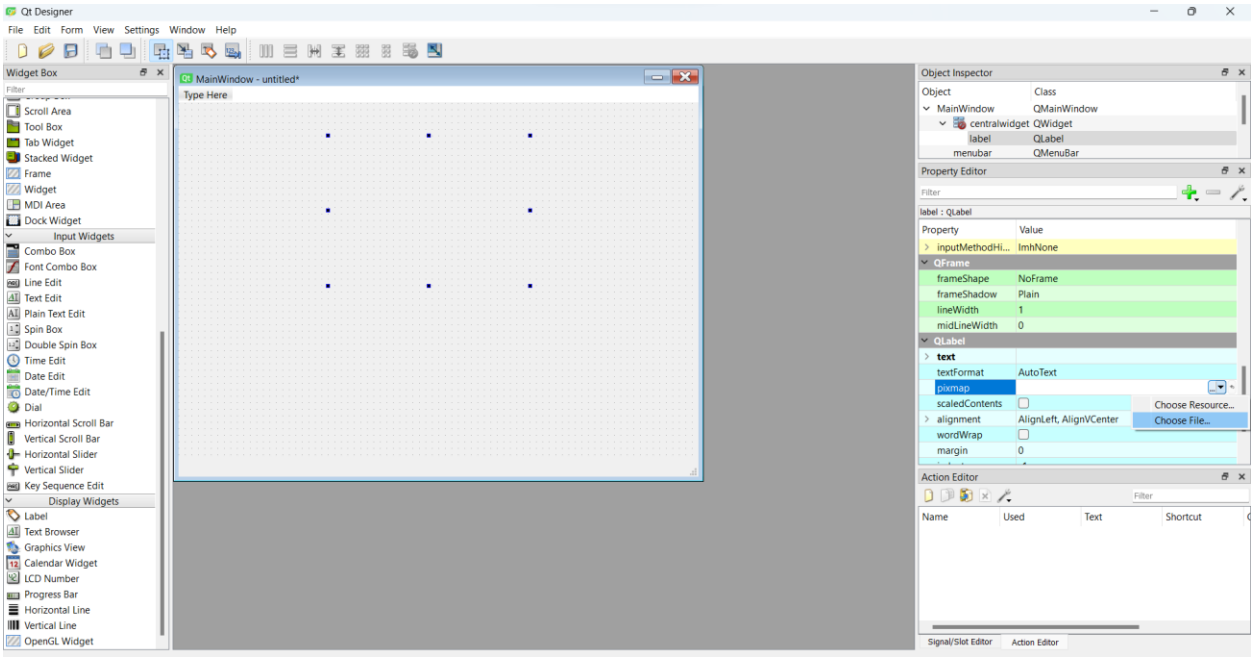
ComboBox-დან არჩეული ტექსტის წვდომა შესაძლებელია `currentText` ბრძანებით
`self.comboX.currentText()`, მიღებული შედეგი იქნება str ტიპის

ღილაკის დაკავშირების შემდეგ `self.submit.clicked.connect(self.exorr)` გაიშვება `exorr` მეთოდი

```
def exorr(self):
    x = int(self.comboX.currentText())
    y = int(self.comboY.currentText())
    exor = x^y
    self.label.setText("X ^ Y =  " + str(exor))
    print(exor)
```

2. სურათები, popup ფანჯრები, მესიჯ ბოქსები, მენიუები

შესაძლებელია სურათების ატვირთვა, რისთვისაც უნდა გავხსნათ სასურველი ზომის Label, ანუ ტექსტური ჭდე, და pixmap-ზე ორჯერ დაწკაპუნების შემდეგ შევარჩიოთ ფაილი. შეგვიძლია ავტვირთოთ JPG ფორმატის სურათი.

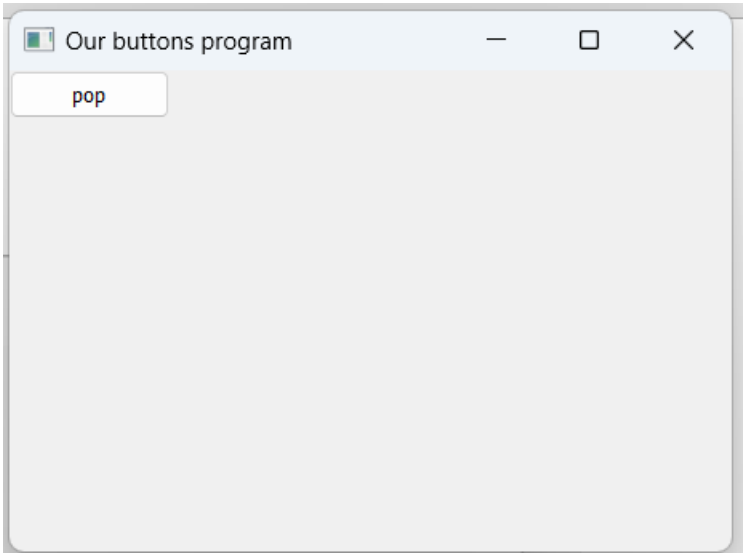


პროგრამის გაშვების შემდეგ სურათი გამოჩნდება ტექსტურ ჭდეში

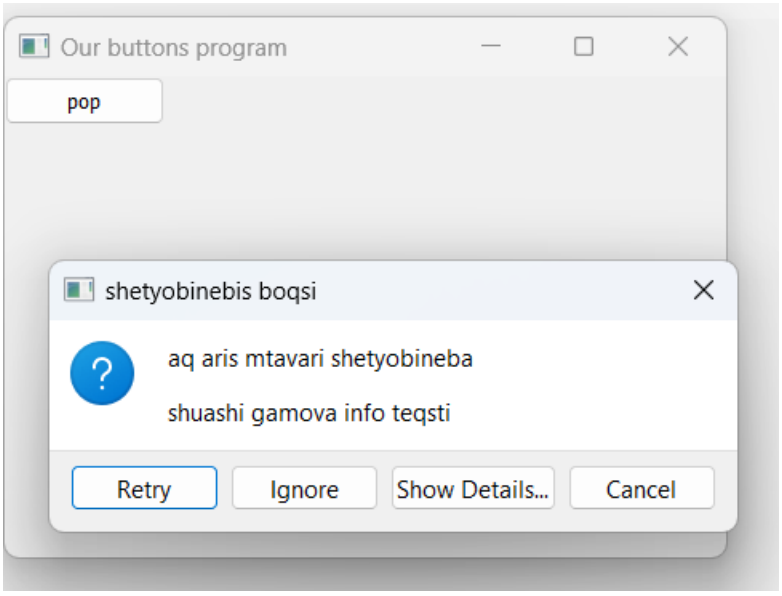
popup ფანჯრების გამოსაყვანად შეგვიძლია გამოვიყენოთ QMessageBox. შევქმნათ მეთოდი show_popup

```
def show_popup(self):
    msg=QMessageBox()
    # msg.setGeometry(100, 100, 100, 100)
    msg.setWindowTitle("shetyobinebis boqsi")
    msg.setText("aq aris mtavari shetyobineba")
    msg.setIcon(QMessageBox.Question)
```

```
        msg.setStandardButtons(QMessageBox.Cancel|QMessageBox.Retry|QMessageBox.I
gnore)
        msg.setDefaultButton(QMessageBox.Retry)
        msg.setInformativeText("shuashi gamova info teqsti")
        msg.setDetailedText("es detalebshi gamova")
        msg.buttonClicked.connect(self.popup_button)
  x=msg.exec_()
```



pop ღილაკზე დაწკაპუნებისას გაიშვება show_popup მეთოდი და გამოვა ფანჯარა
`self.pop.clicked.connect(self.show_popup)`



მესიჯ ბოქსები იქმნება QMessageBox კლასის გამოყენებით

ცალკე ყურადღების ღირსია QMessageBox სტანდარტული ღილაკები
`msg.setStandardButtons(QMessageBox.Cancel|QMessageBox.Retry|QMessageBox.Ignore)`

რომელთა გამოძახებაც შესაძლებელია მათზე დაწკაპუნებით, შგვიძლია მათ დამატებით მივაბათ რაიმე მოქმედება

```
msg.buttonClicked.connect(self.popup_button)
```

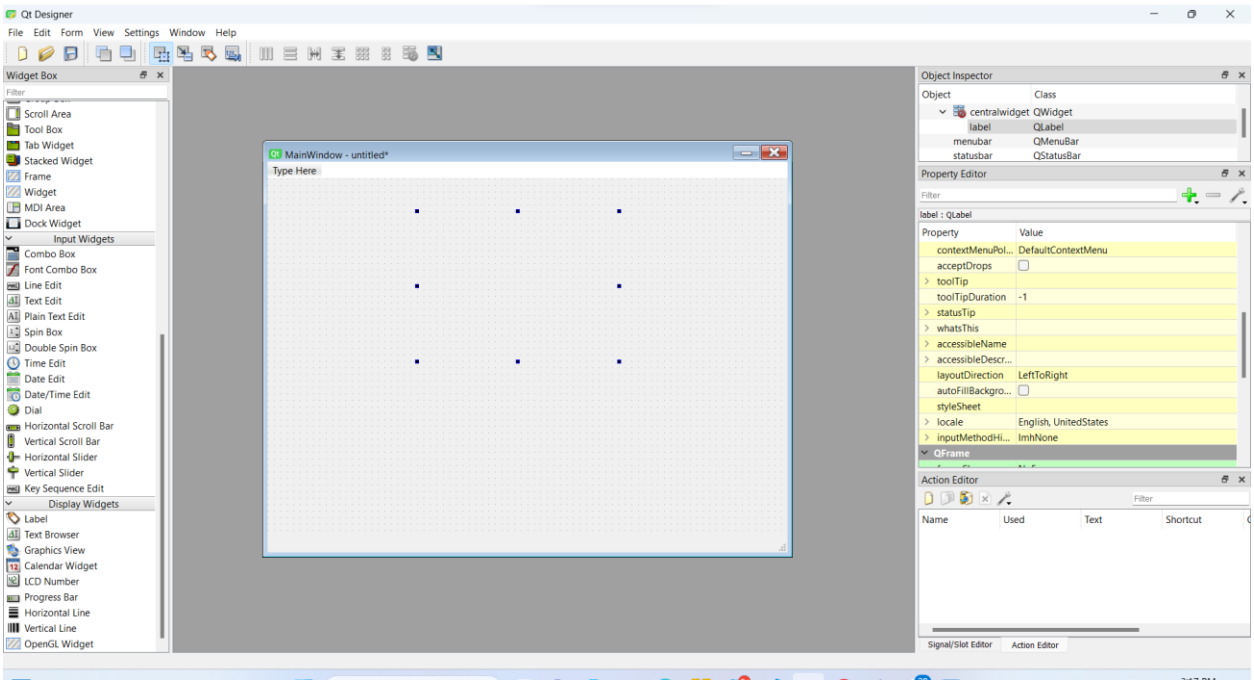
დაწკაპუნებისას დაუკავშირდება `popup_button` მეთოდს

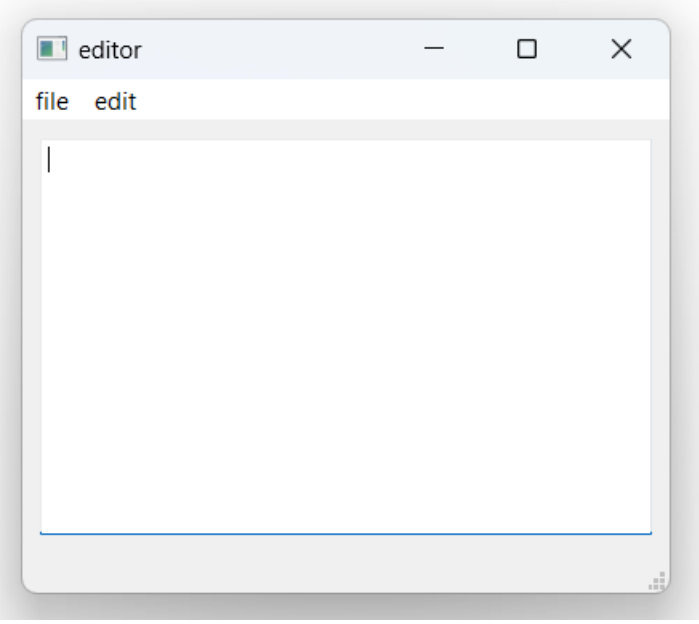
```
def popup_button(self, i):
    print(i.text())
```

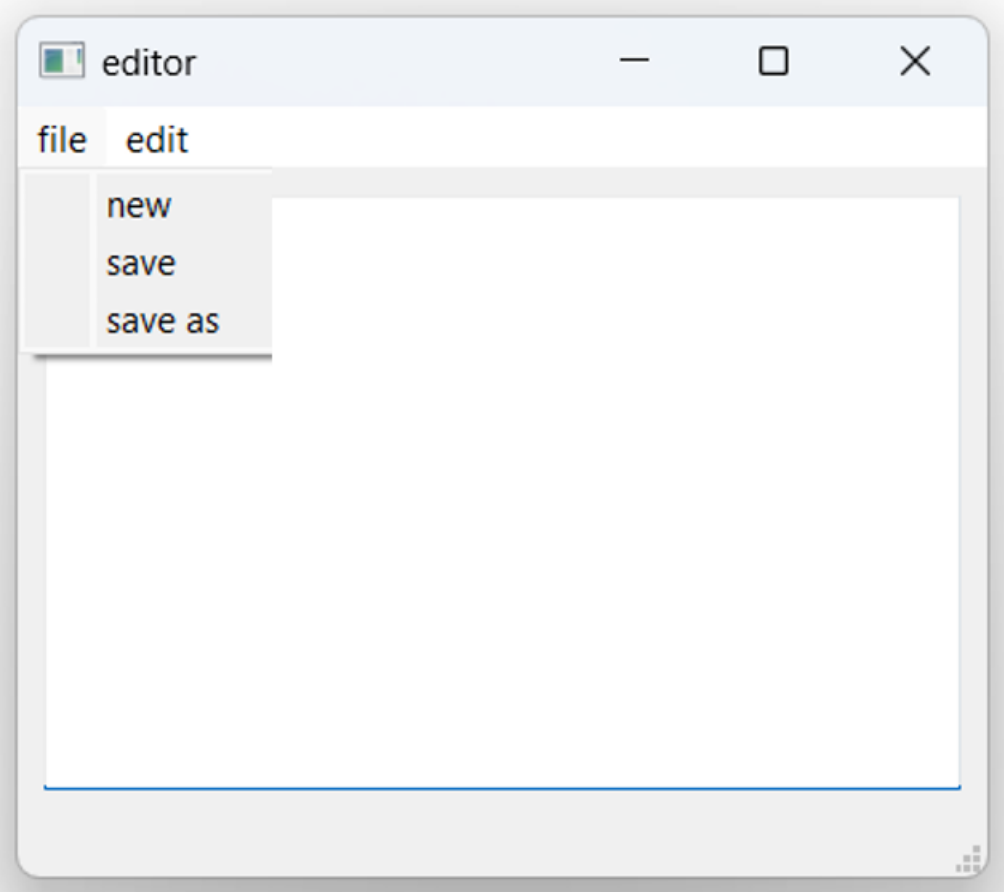
რომელსაც არგუმენტის სახით გადაეცემა დაწკაპუნებული ღილაკი.

მენიუ ბარის შექმნა

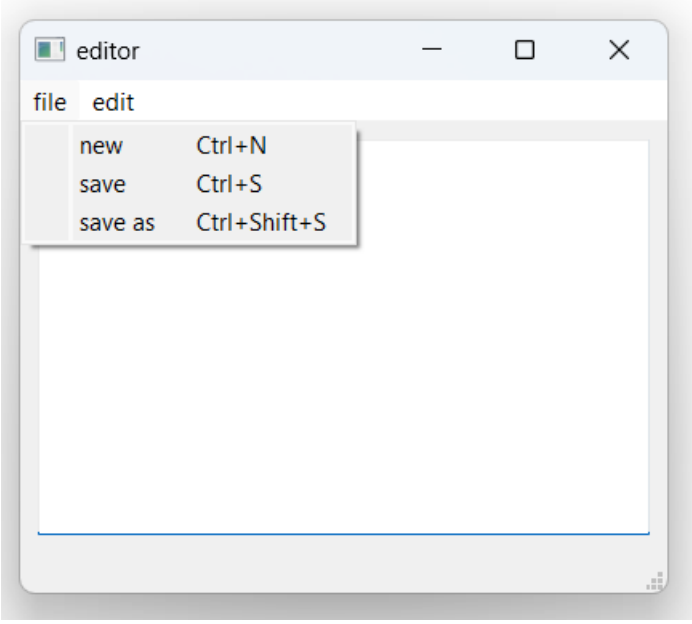
Qt Designer-ში ფანჯრის მენიუ ბარიდან ვირჩევთ Type here მოდულს, მასზე ორჯერ დაწკაპუნების შემდეგ გამოჩნდება კურსორი, აქ შეგვიძლია მენიუს დასახელების ჩაწერა. Enter ღილაკზე დაჭერით შეიქმნება მენიუ. გამოჩნდება ახალი ველი სადაც ანალოგიური მეთოდით შეგვიძლია მენიუს შიგთავსის შექმნა.




editor
file
edit


editor
file
edit
new
save
save as

შგვიძლია თითოეულ ოფციას შევურჩიოთ შესაბამისი Hotkey ღილაკები, ამისთვის უნდა მოვნიშნოთ სასურველი ოფცია, შემდეგ Shortcut მენიუ და კლავიატურაზე დავაჭიროთ ღილაკების სასურველ კომბინაციას.



3. პროექტი გრაფიკული ინტერფეის და კლასების გამოყენებით

შექმენით ფანჯარა, სადაც იქნება შემდეგი ვიჯეტები:
Label - დიდი ზომის, ფანჯრის საერთო ფონისთვის
Label2 - მცირე ზომის, სადაც ჩაჯდება მატარებლის სურათი. უნდა განთავსდეს ფანჯრის მარცხენა ნაწილში
PushButton - რომელიც დააგენერირებს ვაგონების რენდომულ რაოდენობას
Label3 - სადაც გამოჩნდება ვაგონების რაოდენობა
PushButton2 - რომელიც მოძრაობაში მოიყვანს მატარებელს მარცხნიდან მარჯვნივ, მოძრაობას უნდა დაერთოს მატარებლის ხმა.

4. პარალელური პროგრამირება, ნაკადები, პროგრამირება ნაკადების გამოყენებით, threading ბიბლიოთეკა, time ბიბლიოთეკა.

პარალელური პროგრამირება
რისთვის ვიყენებთ პარალელურ პროგრამირებას?
იმისთვის, რომ ავაჩქაროთ ჩვენი პროგრამა, სხვადასხვა ამოცანის ერთდროულად-კონკურენტულად გაშვების გზით, აჩქარება არ არის გარანტირებული, ეს დამოკიდებულია, თუ რისი გაკეთება გვსურს.
დავიწყოთ ყველაზე მარტივი მაგალითიდან და ვნახოთ როგორ მიუშაობს და შედეგ ავაწყოთ უფრო რთული კოდი, შემოგვყავს `time` მოდული და ვქმნით ამთვლელს
`start = time.perf_counter()` იმისათვის, რომ დავთვალოთ რა დრო დაჭირდება ამ კოდის შესრულებას, შემდეგ მოდის ჩვეულებრივი ფუნქცია, რომელიც გვიბეჭდავს რაღცას და აძინებს პროგრამას 2 წამით, როცა ვუშვებთ ფუნქციას, `finish`-ში ვიღებთ პროგრამის

დასრულების დროს, რადგან მასში შეყვანილი მეთოდი ზომავს დროს პროცესორული დროით. შემდეგ იბეჭდება დაწყების და დასრულების დროებს შორის სხვაობა.

```
import time

start = time.perf_counter()

def test():
    print('sleeping 2 seconds')
    time.sleep(2)
    return 'Done Sleeping...}'


test()

finish = time.perf_counter()

print('Finished in', round(finish-start, 2), 'second(s)')
```

გაშვებისას ვნახავთ, რომ ფუნქცია გაიშვა ერთხელ და კოდის სრულად შესრულებას დაჭირდა 2 წამი.
თუ ორჯერ გავუშვებთ ამ ფუნქციას დაჭირდება 4 წამი, რადგან ფუნქციის გაშვების ყოველ ჯერზე აძინებს პროგრამას 2 წამით და ეს ხდება თნმიმდევრობით, ჯერ გაეშვება პირველი ფუნქცია,და მისი დასრულების შემდეგ გაეშვება მეორე ფუნქცია, ამას უწოდებენ სინქრონულ მუშაობას, ამ დროს არაფერი განსაკუთრებული არ ხდება პროცესორში, უბრალოდ ელოდება თანმიმდევრობას.
როგორ უნდა მივიღოთ სარგებელი პარალელური პროგრამირებით?
აქ ორი ვარიანტია არსებობს პროცესორზე მიბმული და IO -ზე მიბმული ამოცანები, პროცესორზე მიბმული დავალებები, ამუშავებენ ბევრ მონაცემებს და იყენებენ პროცესორს, ხოლო IO -ზე მიბმული ამოცანები ელოდებიან შეყვანის და გამომავალი ოპერაციების მთელ რიგს, შესასრულებლად, და პროცესორს ნაკლებად ტვირთავენ, IO დავალებები შიძლება მოიცავდეს ფაილების სისტემიდან სხვადასხვა ფაილების კითხვას და ჩაწერას, სხვადასხვა ფაილების სისტემასთნ დაკავშირებული სხვა ოპერაციებს, ქსელურ ოპერაციებს, ონლაინიდან ფაილების ან სურათების გადმოწერას და სხვა. როცა ვაკეთებთ ე.წ. სრედინგს შგვიძლია სარგებლის მიღება იო ოპერაციებისას, შიძლება ბევრი ფაილი იყოს წასაკითხი ან ქსელური ოპერაცია იყოს შესასრულებელი, ხოლო ამ შმთხვევაში პროცესორზე მიბმული ამოცანები არ მოგვცემს დიდ შედეგებს, შესაძლოა უფრო მეტი დროც დასჭირდეს ვიდრე სინქრონულ მუშაობისას უნდა დასჭირვებოდა.
კოკნურენტული გაშვება, ანუ სრედებით გაშვება არ ნიშნავს, რომ ყველა კოდი ერთდროულად გაიშვება, კოდები გაეშვება ერთმანეთის მიყოლებით, როდესაც გაეშვება პირველი ფუნქცია, მერე ეშვება მეორე, მესამე და ა.შ. ფუნქცია, მაგრამ შეიძლება ისინი ერთდროულად, ანუ პარალელურად მუშობდნენ.
გავაკეთოთ იგივე სრედებით, ამ მიზნით შემოვიყვანოთ მოდული `import threading` მოდული და შევქმნათ სრედები ჩვენი ტესტ ფუნქციისთვის
`t1=threading.Thread()`
ეხლა საჭიროა გადავცეთ `target` , რომელიც იქნება ჩვენი `test` ფუნცია:

`t1=threading.Thread(target=test)` ტარგეტი გადაეცემა ფრჩხილების გარეშე, რადგან გადაეცემა მხოლოდ ფუნქცია, აქ არ ხდება მისი გაშვება.

შვქმნათ 4 სრედი:

```
t1=threading.Thread(target=test)
t2=threading.Thread(target=test)
t3=threading.Thread(target=test)
t4=threading.Thread(target=test)
```

ეხლა თუ გავუშვებთ კოდს, ვნახავთ, რომ ფუნქცია არ გაშვებულა, ამისთვის საჭიროა მათი დასტარტვა:

```
t1.start()
t2.start()
t3.start()
t4.start()
```

ეს კოდი გაუშვებს სრედს, მაგრამ ვნახავთ, რომ ფუნქცია იშვება, მაგრამ მივღებთ ასეთ პასუხს:
sleeping 2 seconds
sleeping 2 seconds
sleeping 2 seconds
sleeping 2 seconds
Finished in: 0.0 second(s)
ფუნქციები ეშვება, მაგრამ მივიღეთ არეული, არა ლოგიკური გამომავალი.
თვიდან დაიბეჭდა რომ ძინავს 2 წამი, მუშობა დაასრულა 0 წამში და და ბოლოს დაიბეჭდა რომ დაასრულა ძილი.
აქ პროგრამას რეალურად კი დასჭირდა 2 წამი, მაგრამ 0 წამი იმიტომ დაწერა, რომ ფუნქციებმა კი დაიწყეს ერთდროულად მუშობა, მაგრამ სანამ სრედებს ეძინა, სკრიპტი მუშობდა კონკურენტულად და ჩავიდა ბოლომდე, ასე, რომ დაბეჭდა `finish = time.perf_counter()` განაცხადი, სანამ ჩვენს პროგრამას ეძინა.
თუ გვინდა, რომ სრედებმა მუშობა დაასრულოს , მანამ , სანამ დავითვლით და დავბეჭდავთ დასრულების დროს, უნდა გამვიყენოთ `join()` მეთოდი.

```
import time
import threading

start = time.perf_counter()

def test():
    print('sleeping 2 seconds')
    time.sleep(2)
    return 'Done Sleeping...}'


t1=threading.Thread(target=test)
t2=threading.Thread(target=test)
t3=threading.Thread(target=test)
t4=threading.Thread(target=test)


t1.start()
```

```
t2.start()
t3.start()
t4.start()
t1.join()
t2.join()
t3.join()
t4.join()

finish = time.perf_counter()

print('Finished in: ', round(finish-start, 2), 'second(s)')
```

აქ სრედები თითქმის ერთდროულად გაიშვება
ეს რა თქმა უნდა ცოტა რთული ჩანს, უფრო მარტივი გზაც ვნახოთ.

რა ხდება თუ ფუნქციის გაშვება დაგვჭირდება ათჯერ, ან მეტჯერ?
ამ კოდის სინქრინულად გაშვებას დასჭირდებოდა 20 წამი, ხოლო სრედებში გაშვებას დაახლოებით 2 წამი

ამისთვის გადავაკეთოთ ჩვენი კოდი:
შევქმნათ ციკლი, რისთვისაც უნდა ავიღოთ ჩვენი სრედი და ჩავსვათ ამ ციკლში:

```
for i in range(10):
    t=threading.Thread(target=test)
    t.start(t)
for thread in threads:
    thread.join()
```

და აქვე გავუშვათ დავსტარტოთ ჩვენი სრედი.

მაგრამ აქვე ვერ გავაკეთებთ მათ დაჯოინებას, რადგან გადაიხლართებიან ერთმანეთში, ვინაიდან შესაძლოა ზოგიერთი სრედი დაჯოინდეს, სანამ სხვები გაეშვება და ა.შ. ამიტომ დაჯოინება უნდა გავაკეთოთ ცალკე ლუპში.

ამისთვის შგვიძლია ყველა სრედი, რომელიც შეიქმნება ჩავსვათ ლისტში, შემდეგ შევქმნათ ახალი ციკლი ამ სრედების დასაჯოინებლად: რომ გავუშვათ ვნახავთ, რომ კოდი შესრულდება დაახლოებით 2 წამში,

ვნახოთ როგორ შეიძლება ტესტ ფუნქციისთვის არგუმენტების გადაცემა, შევცვალოთ ფუნქცია შესაბამისად, ჩავუსვათ პარამეტრი ფრჩხილებში:

ეხლა ჩვენი ფუნქცია ელოდება არგუმენტის მიღებას, არგუმენტების გადაცემა შეგვიძლია არგუმენტების სიის სახით. სდ უნდა გადავცეთ არგუმენტი? -- იქ სადაც შევქმენით სრედი:

```
t=threading.Thread(target=test) აქ პარამეტრად დავამატოთ
```

`t=threading.Thread(target=test, args=5)` ჯერჯერობით გადავცეთ ერთი არგუმენტი 5 წამი და გავუშვათ ფუნქცია, რომელიც შეასრულებს კოდს 5 წამში.

სრული კოდი:

```
import time
import threading

start = time.perf_counter()

def test(wami):
    print("sleeping ", wami," seconds")
    time.sleep(wami)
    print('Done Sleeping...}')

threads = []
for i in range(10):
    t=threading.Thread(target=test, args=[5])
    t.start()
    threads.append(t)

for thread in threads:
    thread.join()


finish = time.perf_counter()

print('Finished in: ', round(finish-start, 2), 'second(s)')
```

---

5. concurrent.futures ბიბლიოთეკა, ნაკადები concurrent.futures მოდულში, პროცესები, პროგრამირება პროცესების გამოყენებით

ვნახოთ კიდევ უფრო მარტივი გზა:

ვნახოთ რას აკეთებს `ThreadPoolExecutor-ი`

გამოვიძახოთ `ThreadPoolExecutor` ამისთვის დაგვჭირდება `concurrent.futures` მოდულის გამოძახება. `ThreadPoolExecutor` -ის გამოყენებისას საჭიროა კონტექსტური მენეჯერის გამოყენება. ანუ იგივე უნდა გავაკეთოთ კონტექსტ მენეჯერის გამოყენებით კონტექსტური მენეჯერები განსაზღვრავენ შესასვლელ და გასასვლელ საფეხურს, რომლებიც ავტომატურად გაეშვება ბლოკში შესვლისა და გამოსვლისას. ფაილის ობიექტები პითონში კონტექსტური მენეჯერის ერთ-ერთი ტიპია და მათი გამოსვლის ნაბიჯი ავტომატურად დახურავს ფაილს.

ჩვენი `threads` სიის ზემოთ გამოვიძახოთ:
`with concurrent.futures.ThreadPoolExecutor as executor:`

თუ ეგზეკუტორით გვინდა რამდენიმე ფუნქციის ერთდროულად გაშვება, უნდა დავურთოთ submit მეთოდი, submit მეთოდი ასრულებს დაგეგმილ ოპერაციებს და აბრუნებს futures ობიექტს

```
    f1=executor.submit()
```

სადაც გადავცემთ ფუნქციას და არგუმენტს:

```
with concurrent.futures.ThreadPoolExecutor as executor:
    f1=executor.submit(test, 2)
     print(f1)
```

ანუ submit მეთოდი დღის წერსიგში აყენებს, გეგმავს test ფუნქციის შესრულებას, და აბრუნებს futures ობიექტს, futures ობიექტი კი აკეთებს ფუნქციის შესრულების ენკაფსულაციას და საშუალებას გვაძლევს შევამოწმოთ, როცა ფუნქცია შერულდება, შედეგად შეგვიძლია ფუნქციის მნიშვნელობების დაბრუნება, რომ შევძლოთ მათი გამოყენება. წინა კოდში ვბეჭდავთ მნიშვნელობებს და არაფერს არ ვაბრუნებთ, მოდით დავაბრუნებინოთ მნიშვნელობები ჩვენს ფუნქციას:

```
def test(sec):
    print('sleeping 2 seconds')
    time.sleep(sec)
    return 'Done Sleeping...}'
```

თუ დაგვჭირდება პასუხის მიღება, უნდა დავბეჭდოთ ფუნქცია

ეს ხდება result() მეთოდის გამოყენებით
კონტექსტურ მენეჯერში ჩავამატოთ:

```
import concurrent.futures
import time

def test(seconds):
    print('sleeping', seconds, 'seconds')
    time.sleep(seconds)
    return 'Done Sleeping...}'

with concurrent.futures.ThreadPoolExecutor() as executor:
    f=executor.submit(test,3)
    print (f.result())
    f1=executor.submit(test,4)
    print (f1.result())
    f2=executor.submit(test,3)
    print (f2.result())

finish = time.perf_counter()

print('Finished in', round(finish-start, 2), 'second(s)')
```

ხოლო თუ გვინდა ბევრი ფუნქციის ერთდროულად გაშვება უნდა გამოვიყენოთ ციკლი ან ლისტ კომპრეჰნშენი ან map-ი:

```
with concurrent.futures.ThreadPoolExecutor() as exedutor:
    months=[30,30, 30,30]
    results=exedutor.map(fib, months)

for result in results:
    print(result)

finish = time.perf_counter()

print('Finished in', round(finish-start, 2), 'second(s)'
```

ეხლა ვნახოთ მულტი პროცესინგი
განსვავდება სრედებისგან, ერთდროულად ეშვება ყველა კოდი
გვაქვს სტანდარტული ბიბლიოთეკა multiprocessing
შემოვიყვანოთ
გვაქვს იგივე ფუნქცია, და გავუშვათ ეს ფუნქცია ორჯერ სხვადასხვა პროცესში

```
process1=multiprocessing.Process(target=test)
process2=multiprocessing.Process(target=test)
```

შევქმენით 2 პროცესის ობიექტი, მაგრამ კოდის გაშვებისას ფუნქცია არ იშვება, ამისთვის საჭიროა დასტარტვა და დაჯოინება

```
import time
import multiprocessing

start = time.perf_counter()

def test():
    print('sleeping 3 seconds')
    time.sleep(2)
    print ('Done Sleeping...}')

p1=multiprocessing.Process(target=test)
p2=multiprocessing.Process(target=test)

p1.start()
p2.start()
p1.join()
p2.join()
```

```
finish = time.perf_counter()

print('Finished in',finish-start, 'second(s)')
```

ეხლა გავაკეთოთ 10 პროცესი ციკლის გამოყენებით

```
import time
import multiprocessing

start = time.perf_counter()

def test():
    print('sleeping 3 seconds')
    time.sleep(2)
    print ('Done Sleeping...}')
if __name__ == '__main__':

    processes=[]
    for i in range(10):
        p=multiprocessing.Process(target=test)
        p.start()
        processes.append(p)

    for i in processes:
        i.join()

finish = time.perf_counter()

print('Finished in',finish-start, 'second(s)')
```

მართლია არ გვაქვს 10 პროცესორი
მაგრამ მაინც მალე გააკეთა, რადგან კომპიუტერს შეუძლია ბირთვებს შორის გადართვა,

მოდით გადავცეთ არგუმენტები

ეხლა ვნახოთ უფრო მარტივი და თნამედროვე გზა:
`ProcessPoolExecutor`-ით

შემოვიყვანოთ ძველი `concurrent.futures`

ჯერჯერობით დავტოვოთ ყველაფერი და შევადაროთ:

```
import time
import concurrent.futures

start = time.perf_counter()
def fib(m):
```

```
    if m<=1:
        return 1
    return fib(m-1)+fib(m-2)

def test(sec):
    print('sleeping ', sec, ' seconds')
    time.sleep(sec)
    return ('Done Sleeping...}')

if __name__ =='__main__':


        with concurrent.futures.ProcessPoolExecutor() as executor:

            t1=executor.submit(test, 2)
            print(t1.result())




finish = time.perf_counter()

print('Finished in',finish-start, 'second(s)')
```

ეხლა ვცადოთ რამდენიმეჯერ გაშვება:

```
t1=executor.submit(test, 2)
            print(t1.result())
```

მაგრამ შეგვიძლია სხვა გზაც გამოვიყენოთ, შევქმნათ სია:

```
import time
import concurrent.futures

start = time.perf_counter()
def fib(m):
    if m<=1:
        return 1
    return fib(m-1)+fib(m-2)

def test(sec):
    print('sleeping ', sec, ' seconds')
    time.sleep(sec)
    return ('Done Sleeping...}')

if __name__ =='__main__':
```

```
        with concurrent.futures.ProcessPoolExecutor() as executor:

            t=[executor.submit(test, 2) for i in range(5)]

            # print(list(t))
            for i in t:
                    print(i)


finish = time.perf_counter()

print('Finished in',finish-start, 'second(s)')
```

ეხლა ვცადოთ სხვადასხვა მნიშვნელობის გადაცემა, ამისთვის შევქმნათ არგუმენტების სია:

```
import time
import concurrent.futures

start = time.perf_counter()
def fib(m):
    if m<=1:
        return 1
    else:
        return fib(m-1)+fib(m-2)

def test(sec):
    print('sleeping ', sec, ' seconds')
    time.sleep(sec)
    return ('Done Sleeping...}')

if __name__ =='__main__':


        with concurrent.futures.ProcessPoolExecutor() as executor:
            tveebi=[32,12,4,30,31,32]
            t=[executor.submit(fib, tve) for tve in tveebi]

            for i in concurrent.futures.as_completed(t):
                 print(i.result())



finish = time.perf_counter()

print('Finished in',finish-start, 'second(s)')
```

გავაკეთოთ იგივე map-ით: გავითვალისწინოთ, რომ map-ით აბრუნებს შედეგებს იმ თანმიმდევრობით, რომლითაც დაისტარტა:

6. პროექტი ნაკადებით

Digit_list = [[rd.randint(1, 200), rd.randint(1, 200), rd.randint(1, 200)] for i in range(10)] გამოიმუშვებს 10 ელემენტიან სიას, სადაც თითოეული ელემენტი არის სამ ელემენტიანი სია.
განსაზღვრეთ ტოლფერდა ტრაპეციის კლასი, რომელიც პარამეტრად მიიღებს ზემოთ მოყვანილი სიის ელემენტს, ანუ კლასს უნდა გადაეცეს რომელიმე სამ ელემენტიანი სია random.choice(). კლასს უნდა ჰქონდეე სამი ატრიბუტი:

ტრაპეციის ფუძეები და სიმაღლე.
კლასს ასევე უნდა ჰქონდეს მეთოდები:
კონსტრუქტორი, __str__() (რომელიც დააბრუნებს ტრაპეციის მონაცემებს),

ტრაპეციის ფართობის გამოსათვლელი,
გადატვირთეთ (გადატვირთვა ნიშნავს მეთოდის გადაწერას) ორი ტრაპეციის ფართობის მიხედვით შედარების (<=, ==) შესაბამისი მეთოდი (__le__, __eq__).

შექმენით ტოლფერდა ტრაპეციის მემკვიდრე მართკუთხედების კლასი, და მართკუთხედების კლასის მემკვიდრე კვადრატების კლასი და თითოეულ მათგანში გადატვირთეთ შესაბამისი მეთოდები.
ტრაპეციის ამოცანაში დაამატეთ სამი ობიექტის შექმნა: ტრაპეციის, მართკუთხედის და კვადრატის, გამოთვალეთ თითეული მათგანის ფართობი. ეს მოქმედება უნდა განხორციელდეს პარალელურ რეჟიმში სხვა და სხვა Thread-ებში და ასევე სხვა და სხვა პროცესებში. შეადარეთ მათი ეფექტურობა.

7. რელაციური მონაცემთა ბაზები, sqlite3 ბიბლიოთეკა, sql მოთხოვნები Python-ში, CREATE TABLE, INSERT, SELECT

და
8. უსაფრთხო მუშაობა SQLite-თან, მონაცემთა ბაზებთან მუშაობა კლასებისა და მოდულების გამოყენებით, UPDATE, DELETE

Python SQLite3 მოდული გამოიყენება SQLite მონაცემთა ბაზის Python-თან ინტეგრაციისთვის. ეს არის სტანდარტიზებული Python DBI API 2.0 და უზრუნველყოფს მარტივ და მოსახერხებელ ინტერფეისს SQLite მონაცემთა ბაზებთან ურთიერთობისთვის. საჭირო არ არის ამ მოდულის ცალკე ინსტალაცია.
SQLite უზრუნველყოფს მსუბუქ, დისკზე დაფუძნებულ მონაცემთა ბაზას, რომელიც არ საჭიროებს ცალკე სერვერულ პროცესს და იძლევა მონაცემთა ბაზაზე წვდომის საშუალებას SQL მოთხოვნებენის არასტანდარტული ვარიანტის გამოყენებით. ზოგიერთ აპლიკაციას SQLite შეუძლია გამოიყენოს მონაცემთა შესანახად. ასევე შესაძლებელია აპლიკაციის პროტოტიპის შექმნა SQLite-ის გამოყენებით და შემდეგ კოდის გადატანა უფრო დიდ მონაცემთა ბაზაში, როგორიცაა PostgreSQL ან Oracle.

პირველ რიგში, უნდა შევქმნათ ახალი მონაცემთა ბაზა და გავხსნათ მონაცემთა ბაზის კავშირი, რათა sqlite3-ს მივცეთ მასთან მუშაობის საშუალება. გამოვიძახოთ sqlite3.connect() მონაცემთა ბაზის `employ`.db-თან კავშირის შესაქმნელად მიმდინარე სამუშაო დირექტორიაში:

```
import sqlite3
conn = sqlite3.connect("employ.db")
```
დაბრუნებული კავშირის ობიექტი conn წარმოადგენს კავშირს დისკზე მონაცემთა ბაზასთან.

იმისათვის, რომ შევასრულოთ SQL განცხადებები და მივიღოთ შედეგები SQL მოთხოვნებიდან, დაგვჭირდება მონაცემთა ბაზის კურსორის გამოყენება. გამოვიძახოთ con.cursor() კურსორის შესაქმნელად:
ახლა, როდესაც ჩვენ მივიღეთ მონაცემთა ბაზასთან კავშირი და კურსორი, შეგვიძლია შევქმნათ მონაცემთა ბაზის ცხრილი, სვეტებით სათაურის მიხედვით. სიმარტივისთვის, შეგვიძლია გამოვიყენოთ სვეტების სახელები ცხრილის დეკლარაციაში. შევასრულოთ CREATE TABLE განცხადება c.execute(...):
```
c.execute("""CREATE TABLE employees (
          first text,
          last text,
          pay integer
          )""")
```

ვნახავთ, რომ შეიქმნა ახალი ცხრილი SQLite-ში ჩაშენებული sqlite_master ცხრილის მოთხოვნით, რომელიც ახლა უკვე უნდა შეიცავდეს ჩანაწერს 'employees' დასათურებით. შევასრულოთ ეს მოთხოვნა c.execute(...) გამოძახებით, მივანიჭოთ შედეგი res და გამოვიძახოთ res.fetchone() მიღებული მწკრივის მისაღებად:

```
import sqlite3
conn = sqlite3.connect(':memory:')
c = conn.cursor()
c.execute("""CREATE TABLE employees (
          first text,
          last text,
          pay integer
          )""")
res = c.execute("SELECT name FROM sqlite_master")
print(res.fetchone())
```

გამომავალი:
('employees',)
ჩვენ ვხედავთ, რომ ცხრილი შეიქმნა, რადგან მოთხოვნა აბრუნებს ცხრილის სახელის შემცველ კორტეჟს. თუ sqlite_master -ით მოვითხოვთ არარსებული ცხრილის სახელს, მაგ spam-ს, res.fetchone() დააბრუნებს None-ს:
```
res = c.execute("SELECT name FROM sqlite_master WHERE name='spam'")
print(res.fetchone())
```

გამომავალი:
None
ახლა დავამატოთ მონაცემების რამდენიმე მწკრივი, რომლებიც გადაეცემა SQL ლიტერალების სახით INSERT განცხადების შესრულებით, კიდევ ერთხელ c.execute(...):
```
c.execute("""
    INSERT INTO employees VALUES
        ('John', 'Doe', 19000),
        ('Jane', 'Doe', 18000)
""")
```

INSERT განცხადება ხსნის ტრანზაქციას, რომელიც უნდა განხორციელდეს, სანამ ცვლილებები შეინახება მონაცემთა ბაზაში. ტრანზაქციის დასასრულებლად კავშირის-conn ობიექტისთვის უნდა გამოვიძახოთ conn.commit():

```
import sqlite3
import employer
conn = sqlite3.connect(':memory:')
c = conn.cursor()
c.execute("""CREATE TABLE employees (
          first text,
          last text,
          pay integer
          )""")

c.execute("""
    INSERT INTO employees VALUES
        ('John', 'Doe', 19000),
        ('Jane', 'Doe', 18000)
""")


res = c.execute("SELECT name FROM sqlite_master ")
print(res.fetchone())

c.execute("SELECT * FROM employees")

conn.commit()
conn.close()
```

ჩვენ შეგვიძლია შევამოწმოთ, მონაცემები სწორად იყო ჩასმული, თუ არა SELECT მოთხოვნის შესრულებით. გამოვიყენოთ ახლა უკვე ნაცნობი `c.execute(...) res` ცვლადისთვის შედეგის მინიჭებისთვის და გამოვიძახოთ `res.fetchall()` ყველა მიღებული მწკრივის დასაბრუნებლად:

```
res = c.execute("SELECT first FROM employees")
print (res.fetchall())
```

გამომავალი:
[('John',), ('Jane',)]

შედეგი არის ორი კორტეჟისგან შემდგარი სია, თითოეული შეიცავს ამ შესაბამისი სვეტის მნიშვნელობას.
ახლა ჩავამატოთ კიდევ სამი მწკრივი cur.executemany(...):

```
data = [
    ('Zuka', 'Dvali', 90000),
    ('Mariam', 'Cxadadze', 120000),
    ('Sofo', 'Kapanadze', 90000),
]
c.executemany("INSERT INTO employees VALUES(?, ?, ?)", data)
```

ალბათ შეამჩნიეთ, რომ ჩანაცვლების ველები, ე.წ. პლეისჰოლდერებბი {} გამოიყენება მონაცემების მოთხოვნასთან დასაკავშირებლად. პითონის მნიშვნელობების SQL განცხადებებთან დასაკავშირებლად ყოველთვის გამოიყენეთ ჩანაცვლების ველები სტრიქონების ფორმატირების ნაცვლად, რათა თავიდან აიცილოთ SQL ინექციის შეტევები.

ჩვენ შეგვიძლია გადავამოწმოთ, რომ ახალი რიგები ჩაემატა SELECT მოთხოვნის შესრულებით, ამჯერად მოთხოვნის შედეგების განმეორებით:

```
import sqlite3
conn = sqlite3.connect(':memory:')
c = conn.cursor()
c.execute("""CREATE TABLE employees (
          first text,
          last text,
          pay integer
          )""")

c.execute("""
    INSERT INTO employees VALUES
        ('John', 'Doe', 19000),
        ('Jane', 'Doe', 18000)
""")

data = [
    ('Zuka', 'Dvali', 90000),
    ('Mariam', 'Cxadadze', 120000),
    ('Sofo', 'Kapanadze', 90000),
]
c.executemany("INSERT INTO employees VALUES(?, ?, ?)", data)

conn.commit()  # არ დაგავიწყდეთ, commit -ის გმოყენება ტრანზაქციის შესასრულებლად INSERT-ის შესრულების შემდეგ.

print("-----------")

res = c.execute("SELECT name FROM sqlite_master")
print(res.fetchone())

c.execute("SELECT * FROM employees")

print("-----------")

res = c.execute("SELECT first FROM employees")

print (res.fetchall())


conn.commit()
conn.close()
```

შგვიძლია შემდეგი სახის ძებნის განხორციელება:

```
for row in c.execute("SELECT first, pay FROM employees ORDER BY first"):
    print(row)
```

თითოეული მწკრივი არის ორ ელემენტიანი კორტეჟი (სახელი, ხელფასი), რომელიც შეესაბამება მოთხოვნაში არჩეულ სვეტებს.

დაბოლოს, შევამოწმოთ, რომ მონაცემთა ბაზა ჩაიწერა დისკზე, გამოძახებით con.close() არსებული კავშირის დახურვით, ახლის გახსნით, ახალი კურსორის შექმნით და შემდეგ მონაცემთა ბაზის მოთხოვნით:

```
new_con = sqlite3.connect("good11")
new_cur = new_con.cursor()
res = new_cur.execute("SELECT first, pay FROM employees ORDER BY pay DESC")
first, pay = res.fetchone()
print(f'The highest pay employees is {first!s}, in {pay}')
```

შევქმენით SQLite მონაცემთა ბაზა sqlite3 მოდულის გამოყენებით, ჩავწერეთ მონაცემები და დავბეჭდეთ მნიშვნელობები მისგან სხვა და სხვა გზით.
საბოლოო კოდი:

```
import sqlite3
conn = sqlite3.connect('good111')
c = conn.cursor()
c.execute("""CREATE TABLE employees (
          first text,
          last text,
          pay integer
          )""")

c.execute("""
    INSERT INTO employees VALUES
        ('John', 'Doe', 19000),
        ('Jane', 'Doe', 18000)
""")

data = [
    ('Zuka', 'Dvali', 90000),
    ('Mariam', 'Cxadadze', 120000),
    ('Sofo', 'Kapanadze', 90000),
]
c.executemany("INSERT INTO employees VALUES(?, ?, ?)", data)

conn.commit()  # არ დაგავიწყდეთ, commit -ის გმოყენება ტრანზაქციის
დასასრულებლად INSERT-ის შესრულების შემდეგ.

print("-----------")

for row in c.execute("SELECT first, pay FROM employees ORDER BY first"):
    print(row)
```

```
conn.commit()
conn.close()
```

```
new_con = sqlite3.connect("good111") # ვამოწმებთ შექმნილ მონაცემთა ბაზას
new_cur = new_con.cursor()
res = new_cur.execute("SELECT first, pay FROM employees ORDER BY pay DESC")
first, pay = res.fetchone()
print(f'The highest pay employees is {first!s}, in {pay}')
```

commit ()
ნებისმიერი მომლოდინე ტრანზაქციის მონაცემთა ბაზაში შეყვანა. თუ ტრანზაქცია არ არის გახსნილი ეს მეთოდი არის იმუშვებს.

close()
მონაცემთა ბაზის კავშირის დახურვა. ნებისმიერი მომლოდინე ტრანზაქცია არ ხორციელდება პირდაპირ; commit()-ით დახურვამდე დარწმუნდით, რომ ყველა ტრანზაქცია შესრულდა, რათა არ დაკარგოთ მომლოდინე ცვლილებები.

მონაცემების ჩანაცვლება, ჩმატება წაშლა შესაძლებელია შესაბამისი მოთხოვნებით. სრული კოდი:

```
import sqlite3
import employer
conn = sqlite3.connect(':memory:')
c = conn.cursor()
c.execute("""CREATE TABLE employees (
            first text,
            last text,
            pay integer
            )""")

def add_emp(employer):
    with conn:
        c.execute("INSERT INTO employees VALUES (:first, :last, :pay)", {'first':
employer.first, 'last': employer.last, 'pay':employer.pay})
def get_emp_name(emp_name):
    c.execute("SELECT * FROM employees WHERE last=:last", {'last':emp_name})
    return c.fetchall()
def update_emp_pay(emp, pay):
    with conn:
        c.execute("""UPDATE employees SET pay = :pay
        WHERE first=:first AND last = :last """,
        {'first':emp.first, 'last':emp.last, 'pay':pay})
def del_emp(emp):
    with conn:
```

```
        c.execute("DELETE from employees WHERE first=:first AND last=:last",
                  {'first': emp.first, 'last': emp.last})

emp_1 = employer.Employee('John', 'Doe', 180000)
emp_2 = employer.Employee('Jane', 'Doe', 190000)
emp_3 = employer.Employee('Zuka', 'Kevanishvili', 90000)
emp_4 = employer.Employee('luka', 'sakandelize', 250000)

add_emp(emp_1)
add_emp(emp_2)
add_emp(emp_3)
add_emp(emp_4)

# c.execute("SELECT * FROM employees")
# print (c.fetchall())

# print(get_emp_name("Doe"))

# update_emp_pay(emp_3, 200000)
# print(get_emp_name('Kevanishvili'))

del_emp(emp_1)
c.execute("SELECT * FROM employees")
print (c.fetchall())
conn.commit()
conn.close()
```

9. არარელაციური მონაცემთა ბაზები, MongoDB, insert_one, insert_many, find, delete_one, delete_many, update_one, update_many

პითონის გამოყენება შესაძლებელია მონაცემთა ბაზის აპლიკაციებში. ერთ-ერთი ყველაზე პოპულარული NoSQL - არარელაციური მონაცემთა ბაზა არის MongoDB.
MongoDB ინახავს მონაცემებს JSON-ის მსგავს დოკუმენტებში, რაც მონაცემთა ბაზას ძალიან მოქნილს და მასშტაბურს ხდის.

უფასო MongoDB მონაცემთა ბაზის ჩამოტვირთვა შესაძლებელია შემდეგ ბმულზე: https://www.mongodb.com. პითონს სჭირდება MongoDB დრაივერი MongoDB მონაცემთა ბაზაზე წვდომისთვის. ჩვენ გამოვიყენებთ MongoDB -ის"PyMongo" დრაივერს. მის დასაყენებლად ვიყენებთ PIP-ს.
pip install pymongo
შემდეგ გვჭირდება მისი ჩართვა ფაილში

```
import pymongo
```

MongoDB მონაცემთა ბაზის შექმნა
MongoDB-ში მონაცემთა ბაზის შექმნა უნდა დავიწყოთ MongoClient ობიექტის შექმნით, შემდეგ უნდა მივუთითოთ კავშირის URL სწორი ip მისამართით და მონაცემთა ბაზის სახელი, რომლის შექმნაც გვსურს.

MongoDB შექმნის მონაცემთა ბაზას, თუ ის არ არსებობს და დაამყარებს კავშირს მასთან.

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")

mydb = myclient["test"]
```

გაითბალისწინეთ: MongoDB-ში მონაცემთა ბაზა არ იქმნება მანამ, სანამ ის არ მიიღებს მონაცემებს!
MongoDB ელოდება სანამ არ შექმნით კოლექციას (ცხრილს), მინიმუმ ერთი ჩანაწერით.
იმისათვის, რომ გავიგოთ არსებოს თუ არა მონაცემთა ბაზა, უნდა დავბეჭდოთ არსებული მონაცემთ ბაზების სია:

```
print(myclient.list_database_names())
```

ან შევამოწმოთ კონკრეტული მონაცემთა ბაზა სახელით:

```
dblist = myclient.list_database_names()
if "test" in dblist:
  print("The database exists.")
```

სრული კოდი:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")

mydb = myclient["test"]

# print(myclient.list_database_names())

dblist = myclient.list_database_names()
if "test" in dblist:
  print("The database exists.")
```

MongoDB კოლექციის შექმნა
კოლექცია MongoDB-ში იგივეა, რაც ცხრილი SQL მონაცემთა ბაზებში. მის შესაქმნელად უნდა გამოვიყენოთ მონაცემთა ბაზის ობიექტი და მივუთითოთ კოლექციის სახელი, რომლის შექმნაც გვსურს. MongoDB შექმნის კოლექციას, თუ ის არ არსებობს.

შევქმნათ კოლექცია 'customers'

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]

mycol = mydb["customers"]
```

მონაცემთ ბაზაში კოლექციის არსებობის შემოწმება:

```
print(mydb.list_collection_names())
```

ასევე შგვიძლია კოლექციის შემოწმება სახელის მიხედვით:

```
collist = mydb.list_collection_names()
if "customers" in collist:
  print("The collection exists.")
```

MongoDB ელოდება დოკუმენტის მიღებას, სანამ ის რეალურად შექმნის კოლექციას. გაითვალისწინეთ: MongoDB-ში კოლექცია არ იქმნება მანამ, სანამ ის არ მიიღებს კონტენტს, ასე რომ, თუ პირველად ქმნით კოლექციას, უნდა შიყვანოთ კონტენტი, სანამ შეამოწმებთ, არსებობს თუ არა კოლექცია!

Insert
ჩანაწერის, ან დოკუმენტის, როგორც მას MongoDB-ში უწოდებენ, კოლექციაში ჩასასმელად, ვიყენებთ insert_one() მეთოდს. (დოკუმენტი MongoDB-ში იგივეა, რაც ჩანაწერი SQL მონაცემთა ბაზებში.)

insert_one() მეთოდის პირველი პარამეტრი არის ლექსიკონი, რომელიც შეიცავს დოკუმენტის თითოეული ველის სახელ(ებ)ს და მნიშვნელობა(ებ)ს, რომლის ჩასმაც გსურთ.
```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["test"]
mycol = mydb["customers"]

mydict = { "name": "Homer", "Simpson": "Springfield 37" }

x = mycol.insert_one(mydict)
```

კოლექციის დაბეჭდვა უკვე მოგვცემს შედეგს:
```
print(mydb.list_collection_names())
```

გამომავალი:
['customers']

_id ველი
მეთოდი insert_one() აბრუნებს InsertOneResult ობიექტს, რომელსაც აქვს თვისება, inserted_id, რომელიც შეიცავს ჩასმული დოკუმენტის id-ნომერს. მაგ. ჩავსვათ კიდევ ერთი ჩანაწერი "customers" კოლექციაში და დავაბრუნოთ _id ველის მნიშვნელობა:
```
# import pymongo

# myclient = pymongo.MongoClient("mongodb://localhost:27017/")

# mydb = myclient["test"]

# print(myclient.list_database_names())

# dblist = myclient.list_database_names()
# if "test" in dblist:
#   print("The database exists.")
```

```
# mycol = mydb["customers"]
# print(mydb.list_collection_names())

# collist = mydb.list_collection_names()
# if "customers" in collist:
#   print("The collection exists.")

import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["test"]
mycol = mydb["customers"]

mydict = { "name": "Hommer", "Simpson": "742 Evergreen Terrace" }

x = mycol.insert_one(mydict)
print(mydb.list_collection_names())

mydict = { "name": "Nelson", "address": "430 Spalding Way" }

x = mycol.insert_one(mydict)

print(x.inserted_id)
```

თუ არ მივუთითებთ _id ველს, მაშინ MongoDB დაამატებთ ერთს მონაცემს და მიანიჭებს უნიკალურ id-ს თითოეული დოკუმენტს. ზემოთ მოყვანილ მაგალითში არ იყო მითითებული _id ველი, ამიტომ MongoDB-მ შექმნა და მიანიჭა უნიკალური _id ჩანაწერს (დოკუმენტს).

მრავალი დოკუმენტის ჩასმა

MongoDB-ში კრებულში მრავალი დოკუმენტის ჩასასმელად, ვიყენებთ insert_many() მეთოდს. insert_many() მეთოდის პირველი პარამეტრი არის სია, რომელიც შეიცავს ლექსიკონებს იმ მონაცემებით, რომელთა ჩასმაც გვსურს:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["test"]
mycol = mydb["customers"]

mylist = [
  { "name": "Moe", "address": "Apple st 652"},
  { "name": "Hannah", "address": "Mountain 21"},
  { "name": "Lisa", "address": "Valley 345"},
  { "name": "Sandy", "address": "Ocean blvd 2"},
  { "name": "Betty", "address": "Green Grass 1"},
  { "name": "Richard", "address": "Sky st 331"},
  { "name": "Susan", "address": "One way 98"},
  { "name": "Vicky", "address": "Yellow Garden 2"},
  { "name": "Ben", "address": "Park Lane 38"},
  { "name": "William", "address": "Central st 954"},
  { "name": "Chuck", "address": "Main Road 989"},
  { "name": "Viola", "address": "Sideway 1633"}
```

```
]

x = mycol.insert_many(mylist)

# დაბეჭდე ჩასმული დოკუმენტების _id მნიშვნელობების სია:

print(x.inserted_ids)
```

მეთოდი insert_many() აბრუნებს InsertManyResult ობიექტს, რომელსაც აქვს თვისება, inserted_ids, რომელიც შეიცავს ჩასმული დოკუმენტების ID-ნომრებს.

შესაძლებელია მრავალი დოკუმენტის ჩსმა, მითითებული ID-ებით
თუ არ გვსურს MongoDB-ს მიერ ჩვენი დოკუმენტისთვის უნიკალური ID-ების შექმნა და მინიჭება, შეგვიძლია მიუთითოთ _id ველი დოკუმენტ(ებ)ის ჩასმისას.

გახსოვდეთ, რომ მნიშვნელობები უნდა იყოს უნიკალური. ორ დოკუმენტს არ შეიძლება ჰქონდეს ერთი და იგივე ID ნომერი.

Find()
MongoDB-ში ვიყენებთ find() და find_one() მეთოდებს, რათა ვიპოვოთ მონაცემები კოლექციაში. ისევე, როგორც SELECT განცხადება გამოიყენება MySQL მონაცემთა ბაზაში ცხრილში მონაცემების მოსაძებნად.
Find One
MongoDB-ში კოლექციიდან მონაცემების შესარჩევად შეგვიძლია გამოვიყენოთ find_one() მეთოდი. find_one() მეთოდი აბრუნებს პირველივე მოვლენას შერჩევაში.

ვიპოვნოთ პირველი დოკუმენტი customers კოლექციაში:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

x = mycol.find_one()

print(x)
```

Find All
MongoDB-ში ცხრილიდან მონაცემების ასარჩევად, ასევე შეგვიძლია გამოვიყენოთ find() მეთოდი.

find() მეთოდი აბრუნებს შერჩევის ყველა მოვლენას.

find() მეთოდის პირველი პარამეტრი არის მოთხოვნის-query ობიექტი. ამ მაგალითში ჩვენ ვიყენებთ ცარიელ მოთხოვნის ობიექტს, რომელიც აიღებს კოლექციაში არსებულ ყველა დოკუმენტს.

find() მეთოდის პარამეტრების გამრეშე გამოყენება იძლევა იგივე შედეგს, როგორც SELECT * MySQL-ში.

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find():
  print(x)
```

მხოლოდ შერჩეული ველების დაბრუნება
find() მეთოდის მეორე პარამეტრი არის ობიექტი, რომელიც აღწერს რომელი ველები შეიტანოს შედეგში.

ეს პარამეტრი არასავალდებულოა და გამოტოვების შემთხვევაში შედეგში ჩაირთვება ყველა ველი.
მხოლოდ სახელების და მისამართების დაბრუნება და არა _id-ის:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

for x in mycol.find({},{ "_id": 0, "name": 1, "address": 1 }):
  print(x)
```

თქვენ არ გაქვთ უფლება მიუთითოთ 0 და 1 მნიშვნელობები იმავე ობიექტში (გარდა იმ შემთხვევისა, თუ ერთ-ერთი ველი არის _id ველი). თუ თქვენ მიუთითებთ ველს მნიშვნელობით 0, ყველა სხვა ველი მიიღებს მნიშვნელობას 1 და პირიქით:
შენიშვნა: მივიღებთ შეცდომას, თუ ერთსა და იმავე ობიექტში მივუთითებთ 0 და 1 მნიშვნელობებს (გარდა იმ შემთხვევისა, თუ ერთ-ერთი ველი არის _id ველი):

Query შედეგის გაფილტვრა
კოლექციაში დოკუმენტების პოვნისას, შეგიძლიათ შედეგის გაფილტვრა ქუერი ობიექტის გამოყენებით. find() მეთოდის პირველი არგუმენტი არის მოთხოვნის-Query ობიექტი და გამოიყენება ძიების შესაზღუდად.

```
# იპოვნე დოკუმენტები შემდები address-ით "Park Lane 38":

import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Park Lane 38" }
```

```
mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

გაფართოებული მოთხოვნა
გაფართოებული მოთხოვნების შესაქმნელად შესაძლებელია მოდიფიკატორების გამოყენება, როგორც მნიშვნელობები ქუაიერი ობიექტში. მაგალითად. იმ დოკუმენტების საპოვნელად, სადაც "address" ველი იწყება ასო "S" ან ანბანის მიხედვით უფრო მაღალი სიმბოლოთი, შეგვიძლია შემდეგი მოდიფიკატორის გამოყენება: {"$gt": "S"}:

მაგალითი
იპოვეთ დოკუმენტები, სადაც მისამართი იწყება ასო "S" ან უფრო მაღლით:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$gt": "S" } } # myquery = { "address": { "$lt": "S" }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

ფილტრაცია რეგულარული გამოსახულებებით
ასევე შეიძლება რეგულარული გამონათქვამების გამოყენება მოდიფიკატორის სახით. რეგულარული გამონათქვამები შეიძლება გამოყენებულ იქნას მხოლოდ სტრიქონების მოთხოვნისთვის.

მოძებნე მხოლოდ ის დოკუმენტების, სადაც "მისამართის" ველი იწყება ასო "S"-ით, გამოიყენეთ რეგულარული გამოთქმა {"$regex": "^S"}:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }

mydoc = mycol.find(myquery)

for x in mydoc:
  print(x)
```

Sort - დალაგება
გამოიყენეთ sort() მეთოდი, რათა დაალაგოთ შედეგი ზრდადობით ან კლებადობით.

sort() მეთოდი იღებს ერთ პარამეტრს "fieldname" და ერთ პარამეტრს "direction"-ისთვის (აღმავალი არის ნაგულისხმევი მიმართულება).

დაალაგეთ შედეგი ანბანურად სახელის მიხედვით:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name")

for x in mydoc:
  print(x)
```

დახარისხება კლებადობით
კლებადობით დასალაგებლად ვიყენებთ მნიშვნელობა -1, როგორც მეორე პარამეტრს.

sort("name", 1) #აღმავალი
sort("name", -1) #დაღმავალი

მაგალითი
დაალაგეთ შედეგი კლებადობით ანბანურად name-ს მიხედვით:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mydoc = mycol.find().sort("name", -1)
```

```
for x in mydoc:
  print(x)
```

Delete -დოკუმენტის წაშლა
ერთი დოკუმენტის წასაშლელად ვიყენებთ delete_one() მეთოდს. delete_one() მეთოდის პირველი პარამეტრი არის query ობიექტი, რომელიც განსაზღვრავს რომელი დოკუმენტის წაშალოს.

შენიშვნა: თუ მოთხოვნამ აღმოაჩინა ერთზე მეტი დოკუმენტი, წაიშლება მხოლოდ პირველი მათგანი.

წაშალე დოკუმენტი რომლის address არის "Mountain 21":

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Mountain 21" }

mycol.delete_one(myquery)
```

ბევრი დოკუმენტის წაშლა
ერთზე მეტი დოკუმენტის წასაშლელად გამოიყენება delete_many() მეთოდი. delete_many() მეთოდის პირველი პარამეტრი არის query ობიექტი, რომელიც განსაზღვრავს რომელი დოკუმენტების წაშალოს.

მაგალითი
წაშალეთ ყველა დოკუმენტი, სადაც address იწყება ასო S-ით:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": {"$regex": "^S"} }

x = mycol.delete_many(myquery)

print(x.deleted_count, " documents deleted.")
```

ყველა დოკუმენტის წაშლა კოლექციაში
კოლექციაშიყველა დოკუმენტის წასაშლელად, უნდა გადავცეთ ცარიელი query ობიექტი delete_many() მეთოდს:

მაგალითი
წაშალე ყველა დოკუმენტი "customers" კოლექციაში:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

x = mycol.delete_many({})

print(x.deleted_count, " documents deleted.")
```

Drop
კოლექციის წაშლა
ცხრილის, ან კოლექციის წაშლა შესაძლებელია ,drop() მეთოდის გამოყენებით.

წაშალე "collection " კოლექცია:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

mycol.drop()
```

drop() მეთოდი აბრუნებს true-ს, თუ კოლექცია წარმატებით იქნა ამოღებული, და false-ს, თუ კოლექცია არ არსებობს.

Update Collection - კოლექციის განახლება
ჩანაწერის, ან დოკუმენტის განახლება ხდება update_one() მეთოდის გამოყენებით.

update_one() მეთოდის პირველი პარამეტრი არის query ობიექტი, რომელიც განსაზღვრავს რომელი დოკუმენტი განაახლოს.

შენიშვნა: თუ query ერთზე მეტ ჩანაწერს აღმოაჩენს, განახლდება მხოლოდ პირველი მათგანი.

მეორე პარამეტრი არის ობიექტი, რომელიც განსაზღვრავს დოკუმენტის ახალ მნიშვნელობებს.

შეცვალე address "Valley 345"- "Canyon 123"-ით:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": "Valley 345" }
newvalues = { "$set": { "address": "Canyon 123" } }

mycol.update_one(myquery, newvalues)

#print "customers" after the update:
```

```
for x in mycol.find():
  print(x)
```

Update Many ბევრის განახლება
ყველა დოკუმენტის განახლებისთვის, რომელიც აკმაყოფილებს მოთხოვნის კრიტერიუმებს, გამოიყენება update_many() მეთოდი.

მაგალითი
განაახლე ყველა დოკუმენტი, სადაც address იწყება ასო "S"-ით:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myquery = { "address": { "$regex": "^S" } }
newvalues = { "$set": { "name": "Minnie" } }

x = mycol.update_many(myquery, newvalues)

print(x.modified_count, "documents updated.")
```

Limit შედეგის შეზღუდვა
MongoDB-ში შედეგის შესაზღუდად ვიყენებთ limit() მეთოდს.

limit() მეთოდი იღებს ერთ პარამეტრს, რიცხვს, რომელიც განსაზღვრავს რამდენი დოკუმენტი დააბრუნოს.

ჩათვალეთ, რომ გაქვთ "მომხმარებელთა" კოლექცია:

```
{'_id': 1, 'name': 'John', 'address': 'Highway37'}
{'_id': 2, 'name': 'Peter', 'address': 'Lowstreet 27'}
{'_id': 3, 'name': 'Amy', 'address': 'Apple st 652'}
{'_id': 4, 'name': 'Hannah', 'address': 'Mountain 21'}
{'_id': 5, 'name': 'Michael', 'address': 'Valley 345'}
{'_id': 6, 'name': 'Sandy', 'address': 'Ocean blvd 2'}
{'_id': 7, 'name': 'Betty', 'address': 'Green Grass 1'}
{'_id': 8, 'name': 'Richard', 'address': 'Sky st 331'}
{'_id': 9, 'name': 'Susan', 'address': 'One way 98'}
{'_id': 10, 'name': 'Vicky', 'address': 'Yellow Garden 2'}
{'_id': 11, 'name': 'Ben', 'address': 'Park Lane 38'}
{'_id': 12, 'name': 'William', 'address': 'Central st 954'}
{'_id': 13, 'name': 'Chuck', 'address': 'Main Road 989'}
{'_id': 14, 'name': 'Viola', 'address': 'Sideway 1633'}
```

შეზღუდე შედეგი მხოლოდ 5 დოკუმენტის დაბრუნებით:

```
import pymongo

myclient = pymongo.MongoClient("mongodb://localhost:27017/")
mydb = myclient["mydatabase"]
mycol = mydb["customers"]

myresult = mycol.find().limit(5)

#print the result:
for x in myresult:
  print(x)
```

10. პროექტი მონაცემთა ბაზების და GUI-ს გამოყენებით
ლექტორი უზრუნველყოფს ამოცანის პირობას

11. Numpy მასივები, მასივების შექმნა, მონაცემთა ტიპები მასივებში, არითმეტიკული ოპერაციები, Numpy მეთოდები
რა არის NumPy?
NumPy არის პითონის ბიბლიოთეკა, რომელიც გამოიყენება მასივებთან მუშაობისთვის. მას ასევე აქვს წრფივი ალგებრის, ფურიეს ტრანსფორმაციისა და მატრიცების დომენში მუშაობის ფუნქციები.

NumPy შეიქმნა 2005 წელს ტრევის ოლიფანტის მიერ. ეს არის ღია კოდის პროექტი და მისი გამოყენება შეძლია ნებისმიერს.

NumPy ნიშნავს რიცხვით პითონს-"Numerical Python".
რისთვისვიყენებთ NumPy-ს?
პითონში გვაქვს სიები, რომლიც მიზანია მასივებზე მუშობა, მაგრამ სიები ძალიან ნელა მუშაობენ. NumPy მიზნად ისახავს მოგვცეს მასივის ობიექტი, რომელიც 50-ჯერ უფრო სწრაფია ვიდრე ტრადიციული Python სიები. NumPy-ში მასივის ობიექტს ჰქვია ndarray, ის უზრუნველყოფს უამრავ დამხმარე ფუნქციას, რაც ძალიან აადვილებს ndarray-თან მუშაობას.
მასივები ძალიან ხშირად გამოიყენება მონაცემთა მეცნიერებაში, სადაც სიჩქარე და რესურსები ძალიან მნიშვნელოვანია. მონაცემთა მეცნიერება: არის კომპიუტერული მეცნიერების განშტოება, სადაც ჩვენ ვსწავლობთ, როგორ შევინახოთ, გამოვიყენოთ და გავაანალიზოთ მონაცემები მისგან ინფორმაციის მისაღებად.
რატომ არის NumPy სიებზე უფრო სწრაფი?
NumPy მასივები სიებისგან განსხვავებით ინახება მეხსიერების ერთ უწყვეტ ადგილას, ამიტომ პროცესებს ძალიან ეფექტურად შეუძლიათ მათზე წვდომა და მანიპულირება. ეს არის მთავარი მიზეზი, რის გამოც NumPy უფრო სწრაფია ვიდრე სიები. ასევე ის ოპტიმიზებულია CPU-ის უახლეს არქიტექტურებთან მუშაობისთვის.

რომელ ენაზეა დაწერილი NumPy?

NumPy არის პითონის ბიბლიოთეკა და დაწერილია ნაწილობრივ პითონში, მაგრამ ნაწილების უმეტესობა, რომლებიც საჭიროებს სწრაფ გამოთვლას, დაწერილია C ან C++-ში.

სად არის NumPy Codebase?
NumPy-ის საწყისი კოდი მდებარეობს github საცავში https://github.com/numpy/numpy

NumPy-ის ინსტალაცია
თუ სისტემაზე უკვე დაინსტალირებული გაქვთ Python და PIP, მაშინ NumPy-ის ძალიან მარტივია, გაუშვით ბრძანება:

pip install numpy
არსებობს პითონის სხვა და სხვა დისტრიბუტივი, რომელსაც უკვე აქვს დაინსტალირებული NumPy, როგორიცაა Anaconda, Spyder და ა.შ.

NumPy-ის დაინსტალირების შემდეგ, შემოგვყავს იგი ჩვენს აპლიკაციებში Import საკვანძო სიტყვის დამატებით:

```
import numpy
```

ახლა NumPy შემოყვანილი და მზად არის გამოსაყენებლად.

```
import numpy

arr = numpy.array([1, 2, 3, 4, 5])

print(arr)
```

გამომავალი:
[1 2 3 4 5]

NumPy as np
მიღებულია NumPy-ს შემოყვანა np ფსევდონიმით.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

NumPy ვერსიის შემოწმება
ვერსიის სტრიქონი ინახება __version__ ატრიბუტის ქვეშ.

```
import numpy as np

print(np.__version__)
```

NumPy ndarray ობიექტის შექმნა

NumPy ndarray ობიექტის შექმნა შეგვიძლია array() ფუნქციის გამოყენებით.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))
```

ndarray-ის შესაქმნელად, ჩვენ შეგვიძლია გამოვიყენოთ სია, კორტეჟი ან ნებისმიერი მასივის მსგავსი ობიექტი array() მეთოდში და ის გარდაიქმნება ndarray-ში:

კორტეჟის გამოყენება NumPy მასივის შესაქმნელად:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

განზომილება მასივებში
მასივების განზომილება არის მასივის სიღრმის ერთი დონე (ჩადგმული მასივები).

ჩადგმული მასივი: არის მასივები, რომლელთა ელემენტები თვითონ მასივებია.

0-D მასივები
0-D მასივები, ან სკალარები, არის მასივის ელემენტები. მასივის თითოეული მნიშვნელობა არის 0-D მასივი.

მაგალითი
შექმენით 0-D მასივი მნიშვნელობით 42

```
import numpy as np

arr = np.array(42)

print(arr)
```

1-D მასივები

მასივს, რომელსაც აქვს 0-D მასივები, როგორც მისი ელემენტები, ეწოდება ერთგანზომილებიანი ან 1-D მასივი.

ეს არის ყველაზე გავრცელებული და ძირითადი მასივები.

1-D მასივი, რომელიც შეიცავს მნიშვნელობებს 1,2,3,4,5:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

2-D მასივები
მასივს, რომელსაც აქვს 1-D მასივები, როგორც მისი ელემენტები, ეწოდება 2-D მასივი. ისინი ხშირად გამოიყენება მატრიცის ან მე-2 რიგის ტენსორების წარმოსაჩენად.

NumPy-ს აქვს მთელი ქვემოდული, რომელიც ეძღვნება მატრიცულ ოპერაციებს, სახელწოდებით numpy.mat

2-D მასივი, რომელიც შეიცავს ორ მასივს 1,2,3 და 4,5,6 მნიშვნელობებით:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)
```

3-D მასივები
მასივს, რომელსაც აქვს 2-D მასივები (მატრიცები), როგორც მისი ელემენტები, ეწოდება 3-D მასივი.

ისინი ხშირად გამოიყენება მე -3 რიგის ტენზორის წარმოსაჩენად.

3-D მასივი ორი 2-D მასივით, ორივე შეიცავს ორ მასივს 1,2,3 და 4,5,6 მნიშვნელობებით:

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(arr)
```

როგორ შევამოწმოთ განზომილება?
NumPy Arrays უზრუნველყოფს ndim ატრიბუტს, რომელიც აბრუნებს მთელ რიცხვს, რომელიც გვეუბნება რამდენი განზომილება აქვს მასივს.

ვამოწმებთ რამდენი განზომილება აქვს მასივებს:

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

უფრო მაღალ განზომილებიანი მასივები
მასივი შეიძლება ჰქონდეს ნებისმიერი განზომილება.

როდესაც მასივი იქმნება, შეგიძლიათ განზომილებების რაოდენობა განსაზღვროთ ndmin არგუმენტის გამოყენებით.

შევქმენით მასივი 5 განზომილებით და ვამოწმებთ, რომ მას აქვს 5 განზომილება:

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('number of dimensions :', arr.ndim)
```

NumPy მასივის ინდექსირება
მასივის ინდექსირება იგივეა რაც მასივის ელემენტზე წვდომა.
მასივის ელემენტზე წვდომა შეგიძლიათ მისი ინდექსის ნომრის მითითებით.
NumPy მასივების ინდექსები იწყება 0-ით, რაც იმას ნიშნავს, რომ პირველ ელემენტს აქვს ინდექსი 0, ხოლო მეორეს აქვს ინდექსი 1 და ა.შ.

მივიღეთ პირველი ელემენტი შემდეგი მასივიდან:

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[0])
```

```
# მივიღეთ მეორე ელემენტი შემდეგი მასივიდან:
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[1])
```

```
# მიიღეთ მესამე და მეოთხე ელემენტები შემდეგი მასივიდან და შვკრიბეთ ისინი.
```

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr[2] + arr[3])
```

წვდომა 2-D მასივებზე
ორგანზომილებიანი მასივიდან ელემენტებზე წვდომისთვის შეგვიძლია გამოვიყენოთ მძიმით გამოყოფილი მთელი რიცხვები, რომლებიც წარმოადგენს ელემენტის განზომილებას და ინდექსს.

წარმოიდგინეთ ორგანზომილებიანი მასივები, როგორიცაა ცხრილი რიგებით და სვეტებით, სადაც განზომილება წარმოადგენს რიგს და ინდექსი წარმოადგენს სვეტს.

მაგალითი
პირველი რიგის და მეორე სვეტის ელემენტზე წვდომა:

```
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[0, 1])
```

```
# მეორე რიგის და მეხუთე სვეტის ელემენტზე წვდომა:
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
```

```
print('5th element on 2nd row: ', arr[1, 4])
```

წვდომა 3-D მასივებზე
3-D მასივის ელემენტებზე წვდომისთვის შეგვიძლია გამოვიყენოთ მძიმით გამოყოფილი მთელი რიცხვები, რომლებიც წარმოადგენს ზომებს და ელემენტის ინდექსს

```
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

print(arr[0, 1, 2])
```

მაგალითის ახსნა:
arr[0, 1, 2] ბეჭდავს მნიშვნელობას 6.

რადგან პირველი რიცხვი წარმოადგენს პირველ განზომილებას, რომელიც შეიცავს ორ მასივს:
[[1, 2, 3], [4, 5, 6]] და: [[7, 8, 9], [10, 11, 12]]

მას შემდეგ, რაც ჩვენ ავირჩიეთ 0, დაგვრჩება პირველი მასივი:
[[1, 2, 3], [4, 5, 6]]

მეორე რიცხვი წარმოადგენს მეორე განზომილებას, რომელიც ასევე შეიცავს ორ მასივს:
[1, 2, 3]
და:
[4, 5, 6]
მას შემდეგ, რაც ჩვენ ავირჩიეთ 1, დაგვრჩება მეორე მასივი:
[4, 5, 6]

მესამე რიცხვი წარმოადგენს მესამე განზომილებას, რომელიც შეიცავს სამ მნიშვნელობას:
4
5
6
მას შემდეგ, რაც ჩვენ ავირჩიეთ 2, მივიღებთ მესამე მნიშვნელობას:
6

ნეგატიური ინდექსირება
უარყოფით ინდექსირებას ვიყენებთ მასივის ბოლოდან წვდომისთვის.

```
# ნეგატიური ინდექსირება
# ბოლო ელემენტის ბეჭდვა მე-2 განზომილებიდან:
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('Last element from 2nd dim: ', arr[1, -1])
```

სლაისინგი
პითონში დაჭრა ნიშნავს ელემენტების ამოღებას ერთი მოცემული ინდექსიდან მეორე მოცემულ ინდექსამდე.

ინდექსის ნაცვლად გადავცემთ სლაისს შემდეგი სახით: [დაწყება:დასრულება]. # start: stop: step

ჩვენ ასევე შეგვიძლია განვსაზღვროთ ბიჯი: [დაწყება: დასრულება: ბიჯი].

თუ არ გადავცემთ არგუმენტს, start იქნება 0

თუ არ გადავცემთ არგუმენტს, stop იქნება სიგრძე მასივის ამ განზომილებაში

თუ არ გადავცემთ step, იქნება 1

შემდეგი მასივიდან ამოვიღეთ ელემენტები 1-დან ინდექს 5-მდე:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

შენიშვნა: შედეგი მოიცავს დაწყების ინდექსს, მაგრამ გამორიცხავს დასრულების ინდექსს.

ამოჭრის ელემენტებს 4 ინდექსიდან მასივის ბოლომდე:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

უარყოფითი ჭრა
მინუს ოპერატორს ვიყენებთ ინდექსზე მითითებისთვის ბოლოდან:

-3 ინდექსიდან ბოლოდან ბოლოდან პირველ ინდექსამდე:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-3:-1])
```

STEP-ბიჯი
step გამოიყენება ბიჯის დასადგენად:

```
# დააბრუნე ყოველი მეორე ელემენტი 1-დან ინდექს 5-მდე:
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
print("----------------------------")
# დააბრუნე ყოველი მეორე ელემენტი მთელ ერეიში:

import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[::2])



# მეორე ელემენტიდან ამოიღე ელემენტები 1-დან ინდექს 4-მდე (არ შედის):
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])


# ორივე ელემენტიდან, ამოჭერი 1-დან 4 ინდექსამდე (არ შედის), დააბრუნებს 2-D
მასივს:
```

```python
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 1:4])
```

NumPy მონაცემთა ტიპები
მონაცემთა ტიპები პითონში
ნაგულისხმევად პითონს აქვს მონაცემთა შემდეგი ტიპები:

სტრიქონები - გამოიყენება ტექსტური მონაცემების წარმოსადგენად, ტექსტი მოცემულია ბრჭყალებში.
მთელი რიცხვი - გამოიყენება მთელი რიცხვების წარმოსადგენად. მაგალითად. -1, -2, -3
float - გამოიყენება ნამდვილი რიცხვების წარმოსადგენად. მაგალითად. 1.2, 42.42
ლოგიკური - გამოიყენება True ან False-ის წარმოსადგენად.
კომპლექსი - გამოიყენება რთული რიცხვების წარმოსადგენად. მაგალითად. 1.0 + 2.0j, 1.5 + 2.5j
მონაცემთა ტიპები NumPy-ში
NumPy-ს აქვს რამდენიმე დამატებითი მონაცემთა ტიპი და ეხება მონაცემთა ტიპებს ერთი სიმბოლოთი, როგორიცაა i მთელი რიცხვებისთვის, u მთელი რიცხვებისთვის ნიშნის გარეშე და ა.შ.

ქვემოთ მოცემულია მონაცემთა ყველა ტიპის სია NumPy-ში და მათი წარმოსადგენად გამოყენებული სიმბოლოები.

i - integer
b - boolean
u - unsigned integer
f - float
c - complex float
m - timedelta
M - datetime
O - object
S - string
U - unicode string
V - fixed chunk of memory for other type (void)

NumPy მასივის ობიექტს აქვს თვისება სახელად dtype, რომელიც აბრუნებს მასივის მონაცემთა ტიპს:

მივიღოთ მასივის ობიექტის მონაცემთა ტიპი:

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr.dtype)
```

```
import numpy as np

arr = np.array(['apple', 'banana', 'cherry'])

print(arr.dtype)
```

მასივების შექმნა განსაზღვრული მონაცემთა ტიპით
ჩვენ ვიყენებთ array() ფუნქციას მასივების შესაქმნელად, ამ ფუნქციას შეუძლია მიიღოს არგუმენტი: dtype, რომელიც საშუალებას გვაძლევს განვსაზღვროთ მასივის ელემენტების მოსალოდნელი მონაცემთა ტიპი:

```
import numpy as np

arr = np.array([1, 2, 3, 4])

print(arr.dtype)
```

ან

```
import numpy as np

arr = np.array(['apple', 'banana', 'cherry'])

print(arr.dtype)
```

```
# შექმენი სტრიქონების მასივი:

import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)
print(arr.dtype)
```

i, u, f, S და U-სთვის შეგვიძლია ასევე ზომის განსაზღვრაც.

```
# შექმენი მასივი 4 ბაიტიანი მთელი რიცხვებისგან:
```

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)
print(arr.dtype)
```

რა მოხდება, თუ ღირებულების კონვერტაცია შეუძლებელია?
თუ მოცემულია ტიპი, რომელშიც ელემენტების ტრანსლირება შეუძლებელია, NumPy გამოიწვევს ValueError-ს.

```
# არა მთელი რიცხვი, როგორიცაა 'a', არ შეიძლება გადაკეთდეს მთელ რიცხვად:
import numpy as np

arr = np.array(['a', '2', '3'], dtype='i')
```

მონაცემთა ტიპის კონვერტაცია არსებულ მასივებში
არსებული მასივის მონაცემთა ტიპის შეცვლის საუკეთესო გზაა მასივის ასლის გაკეთება astype() მეთოდით.

astype() ფუნქცია ქმნის მასივის ასლს და საშუალებას გაძლევთ, პარამეტრად მიუთითოთ მონაცემთა ტიპი.

მონაცემთა ტიპი შეიძლება განისაზღვროს სტრიქონის გამოყენებით, როგორიცაა 'f' float-ისთვის, 'i' მთელი რიცხვისთვის და ა.შ. ან შეგიძლიათ გამოიყენოთ მონაცემთა ტიპი პირდაპირ, როგორიცაა float float-ისთვის და int მთელი რიცხვისთვის.

```
# შეცვალე მონაცემთა ტიპი float-დან მთელ რიცხვზე, პარამეტრის მნიშვნელობად „i“-ის გამოყენებით:
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

სხვაობა Copy და View-ს შორის
მთავარი განსხვავება Copy და მასივის View შორის არის ის, რომ Copy არის ახალი მასივი, ხოლო View მხოლოდ ორიგინალური მასივის წარმოდგენაა.

Copy ფლობს მონაცემებს და ასლში შეტანილი ნებისმიერი ცვლილება არ იმოქმედებს ორიგინალ მასივზე, ხოლო ორიგინალ მასივში შეტანილი ნებისმიერი ცვლილება გავლენას არ მოახდენს ასლზე.

View არ ფლობს მონაცემებს და მასში შეტანილი ნებისმიერი ცვლილება იმოქმედებს თავდაპირველ მასივზე, ხოლო თავდაპირველ მასივში შეტანილი ნებისმიერი ცვლილება გავლენას მოახდენს View-ზე.

Copy :

გააკეთე ასლი, შეცვალე ორიგინალი მასივი და აჩვენე ორივე მასივი:

ორიგინალ მასივში განხორციელებული ცვლილებები არ უნდა გავრცელდეს ასლზე.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42

print(arr)
print(x)
```

VIEW:
გააკეთე VIEW, შეცვალეთ ორიგინალური მასივი და აჩვენე ორივე მასივი:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42

print(arr)
print(x)
```

თავდაპირველ მასივზე უნდა იმოქმედოს view-ში განხორციელებულმა ცვლილებებმა.

როგორ შევამოწმოთ, ფლობს თუ არა Array მის მონაცემებს
როგორც ზემოთ აღვნიშნეთ, ასლი ფლობს მონაცემებს, view კი არ ფლობს მონაცემებს, მაგრამ როგორ შევამოწმოთ ეს?

ყველა NumPy მასივს აქვს `base` ატრიბუტი, რომელიც აბრუნებს None-ს, თუ მასივი ფლობს მონაცემებს.

წინააღმდეგ შემთხვევაში, `base` ატრიბუტი მიმართვს თავდაპირველ ობიექტს.

დაბეჭდე `base` ატრიბუტის მნიშვნელობა, რათა შევამოწმოთ მასივი ფლობს მის მონაცემებს თუ არა:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

x = arr.copy()
y = arr.view()

print(x.base)
print(y.base)
```

copy აბრუნებს None-ს.
view აბრუნებს თავდაპირველ მასივს.

NumPy მასივი Shape
მასივის ფორმა - მასივის ფორმა არის ელემენტების რაოდენობა თითოეულ განზომილებაში.

NumPy მასივებს აქვთ ატრიბუტი სახელწოდებით shape, რომელიც აბრუნებს კორტეჟს, თითოეულ ინდექსი შეიცავს შესაბამისი ელემენტების რაოდენობას.

```
# ამობეჭდე 2-D მასივის ფორმა:
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)
```

ზემოთ მოყვანილი მაგალითი აბრუნებს (2, 4), რაც ნიშნავს, რომ მასივს აქვს 2 განზომილება, სადაც პირველ განზომილებას აქვს 2 ელემენტი, ხოლო მეორეს აქვს 4.

```
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
```

```
print('shape of array :', arr.shape)
```

გამომავალი:
[[[[[1 2 3 4]]]]]
shape of array : (1, 1, 1, 1, 4)

რას წარმოადგენს მიღებული კორტეჟი?
მთელი რიცხვები მიუთითებს ელემენტების რაოდენობაზე, შესაბამის განზომილებაში.

ზემოთ მოცემულ მაგალითში ინდექს-4-აქვს მნიშვნელობა 4, ასე რომ, შეგვიძლია ვთქვათ, რომ მე-5 (4 + 1-ე) განზომილებას აქვს 4 ელემენტი.

მასივის ფორმა არის თითოეულ განზომილებაში არსებული ელემენტების რაოდენობა.

reshap-ით ჩვენ შეგვიძლია დავამატოთ ან წავშალოთ ზომები ან შევცვალოთ ელემენტების რაოდენობა თითოეულ განზომილებაში.

შეცვალეთ ფორმა 1-D-დან 2-D-ზე
გადააქციეთ შემდეგი 1-D მასივი 12 ელემენტით 2-D მასივად.

ყველაზე გარე განზომილებას ექნება 4 მასივი, თითოეულ მათგანს 3 ელემენტი:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

Reshape 1-D-დან 3-D-ით

გადააკეთე შემდეგი 1-D მასივი 12 ელემენტით 3-D მასივად.

```
# ყველაზე გარე განზომილებას ექნება 2 მასივი, რომელიც შეიცავს 3 მასივს,
თითოეულში 2 ელემენტით:
import numpy as np
```

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

შეგვიძლია თუ არა Reshape-ით ნებისმიერი ფორმის მინიჭება?
ამ შემთხვევაში კი, რამდენადაც ელემენტების რაოდენობა ორივე ფორმაში თანაბარია, უფრო სწორედ მიღებული მასივის ელემენტების რაოდენობა 2*3*2 უდრის პირველი მასივის ელემენტების რაოდენობას 12, ხოლო თუ დავაპირებთ განსხვავებული ზომის მასივის მიღებას, დაგენერირდება შეცდომა.

```
# განსხვავებული ზომის მასივის მიღებისას, დაგენერირდება შეცდომა.
# import numpy as np

# arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

# newarr = arr.reshape(3, 3)

# print(newarr)
```

უცნობი განზომილება
შესაძლებელია გვქონდეს ერთი "უცნობი" განზომილება.

ეს ნიშნავს, რომ თქვენ არ გვჭირდება ზუსტი რიცხვის მითითება ერთ-ერთი განზომილებისთვის reshape მეთოდში. შეგვიძლია ჩავწეროთ -1 მნიშვნელობად და NumPy გამოთვლის ამ რიცხვს ჩვენს ნაცვლად.

```
# გადაიყვანე 1D მასივი 8 ელემენტით 3D მასივში 2x2 ელემენტებით:
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(2, 2, -1)

print(newarr)
```

შენიშვნა: არ შეგვიძლია -1 -ის მითითება ერთზე მეტ განზომილებაში.

Flattening

Flattening ნიშნავს მრავალგანზომილებიანი მასივის 1D მასივად გადაქცევას. ამისათვის ასევე შეგვიძლია reshape(-1)-ის გამოყენება.

```
# გადაიყვანე მასივი 1D მასივში:
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

newarr = arr.reshape(-1)

print(newarr)
```

შენიშვნა: მასივების ფორმის შეცვლისთვის არსებობს უამრავი ფუნქცია flatten-ში, ravel და ასევე ელემენტების გადაწყობისთვის rot90, flip, fliplr, flipud და ა.შ.

NumPy მასივის გადარჩევა-იტერაცია
გადარჩევა ნიშნავს ელემენტების სათითაოდ გავლას.

რამდენადაც საქმე გვაქვს მრავალგანზომილებიან მასივებთან numpy-ში, ამის გაკეთება შეგვიძლია პითონის საბაზისო ციკლის გამოყენებით. თუ ჩვენ გადავარჩევთ 1-D მასივს, ის გაივლის თითოეულ ელემენტს სათითაოდ.

```
# 1-D მასივის ელემენტების გადარჩევა:
import numpy as np

arr = np.array([1, 2, 3])

for x in arr:
  print(x)
```

2-D მასივების გადარჩევა
2-D მასივში ის გაივლის ყველა მასივის ელემენტებს

```
# 2-D მასივების გადარჩევა
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
  print(x)
```

თუ გადავარჩევთ n-D მასივს, ის სათითაოდ გაივლის n-1 განზომილებას.

რეალური მნიშვნელობების, სკალარების დასაბრუნებლად, ჩვენ უნდა გავიმეოროთ მასივები თითოეულ განზომილებაში.

```
# 3-D მასივების გამეორება
# 3-D მასივში ის გაივლის ყველა 2-D მასივს.

import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
  print(x)
```

```
# რეალური მნიშვნელობების, სკალარების დასაბრუნებლად უნდა გადავარჩიოთ მასივები თითოეულ განზომილებაში.
import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

for x in arr:
  for y in x:
    for z in y:
      print(z)
```

მასივების გადარჩევა nditer() გამოყენებით
ფუნქცია nditer() არის დამხმარე ფუნქცია, რომელიც შეიძლება გამოყენებულ იქნას, როგორც ძალიან მარტივი ასევე ძალიან რთული გადარჩევისთვის. ის წყვეტს ზოგიერთ ძირითად საკითხს, რომელსაც ჩვენ ვაწყდებით გადარჩევისას. გადავხედოთ მას მაგალითებით.

იტერაცია თითოეულ სკალარ ელემენტზე
საბაზისო ციკლებში, მასივის ყოველი სკალარის გადარჩევისას, ჩვენ უნდა გამოვიყენოთ n რაოდენობის for ციკლი, რომელთა დაწერა შეიძლება რთული იყოს ძალიან მაღალი განზომილების მქონე მასივებისთვის.

```
# გადაარჩიე შემდეგი 3-D მასივი:
import numpy as np
```

```
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

for x in np.nditer(arr):
  print(x)
```

მონაცემთა სხვადასხვა ტიპებით შემდგარი მასივის გადარჩევა
ჩვენ შეგვიძლია გამოვიყენოთ op_dtypes არგუმენტი და გადავცეთ მას მოსალოდნელი მონაცემთა ტიპი, რათა შევცვალოთ ელემენტების მონაცემთა ტიპი გადარჩევისას.

NumPy არ ცვლის ელემენტის მონაცემთა ტიპს ადგილზე (როდესაც ელემენტი არის მასივში), ამიტომ მას სჭირდება სხვა სივრცე ამ მოქმედების შესასრულებლად, ამ დამატებით სივრცეს ეწოდება ბუფერი და იმისათვის, რომ ჩართოთ იგი nditer()-ში გადავცემთ flags=['buffered'].

```
# გადაარჩიე შემდეგი მასივი სტრიქონის სახით:

import numpy as np

arr = np.array([1, 2, 3])

for x in np.nditer(arr, flags=['buffered'], op_dtypes=['S']):
  print(x)
```

გადარჩევა სხვადასხვა ბიჯით
ჩვენ შეგვიძლია გამოვიყენოთ ფილტრაცია და შემდეგ გადარჩევა.

```
# გადაარჩიე 2D მასივის ყველა სკალარული ელემენტი 1 ელემენტის გამოტოვებით:

import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for x in np.nditer(arr[:, ::2]):
  print(x)
```

ზოგჯერ გადარჩევისას გვჭირდება ელემენტის შესაბამისი ინდექსის მიღება, ndenumerate() მეთოდი შეიძლება გამოყენებულ იქნას ამ გამოყენების შემთხვევებისთვის.

```
# ელემენტის შესაბამისი ინდექსის მიღება,
import numpy as np
```

```
arr = np.array([1, 2, 4])

for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Join-NumPy მასივების შეერთება
შეერთება ნიშნავს ორი ან მეტი მასივის შიგთავსის ერთ მასივში მოთავსებას. SQL-ში ცხრილებს ერთმანეთს ვუერთებდით გასაღების საფუძველზე, ხოლო NumPy-ში მასივებს ვუერთდებით ღერძებით.
ამისთვის concatenate() ფუნქციას გადავცემთ მასივების თანმიმდევრობას, შესაერთებელი მასივების თამნმიმდევრობას ღერძთან ერთად. თუ ღერძი არ არის გადაცემული, ის მიიღება როგორც 0.

```
# 2-D მასივების შეერთება მწკრივების გასწვრივ (ღერძი = 1):

import numpy as np

arr1 = np.array([[1, 2], [3, 4]])

arr2 = np.array([[5, 6], [7, 8]])

arr = np.concatenate((arr1, arr2), axis=1)

print(arr)
```

მასივების შეერთება Stack ფუნქციების გამოყენებით
დაწყობა იგივეა, რაც შეერთება, ერთადერთი განსხვავება ისაა, რომ დაწყობა ხდება ახალი ღერძის-axis გასწვრივ.

ჩვენ შეგვიძლია გავაერთიანოთ ორი 1-D მასივი მეორე ღერძის გასწვრივ, რასაც მოჰყვება მათი ერთმანეთზე დაწყობა.

stack() მეთოდს გადავცემთ მასივების თანმიმდევრობას, რომელიც გვინდა შევაერთოთ ღერძთან ერთად. თუ ღერძი არ არის გადაცემული, ის მიიღება როგორც 0.

Stack მასივების ერთმანეთზე დაწყობა.

```
import numpy as np
```

```
arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np.stack((arr1, arr2), axis=1)

print(arr)
```

იმპორტ

მასივებზე შეგვიძლია შევასრულოთ არითმეტიკული ოპერადციები

აქ ერთი ელემენტი ათწილადია, შედეგად მთელი მასივი გადაკეთდება ათწილადად

ხოლო აქ ერთი პარამეტრი სტრინგია დააკვირდით გამომავალს

```
import numpy
a=numpy.array([2,3,4,5])
print(a*2)
b=numpy.array([2,3,4,5.5])
print(b)
c=numpy.array([2,3,4,'hi'])
print(c)


d=numpy.array([[1,2,3], [4,5,6]])
print(d)

d=numpy.array([[1,2,3], [4,5,6]],  "float")
print(d)

მასივში მიმდევრობის შექმნა
d=numpy.arange(1, 10)
print(d)

d=numpy.arange(1, 10, dtype="float")
```

```
print(d)
```

ნულებისგან შემდგარი მასივის შექმნა

```
d=numpy.zeros(5)
print(d)

d=numpy.zeros([5, 2])
print(d)

d=numpy.zeros([5, 2], dtype="int")
print(d)
```

ერთებისგან შემდგარი მასივის შექმნა

```
d=numpy.ones([3, 2], dtype="int")
print(d)

d=numpy.empty([3, 2], dtype="int")
print(d)
print(d*2)
```

ვნახოთ numpy ფუნქციები:

```
import numpy

print(numpy.linspace(1, 100, 5))
print(numpy.linspace(1, 100, num=5))


print(numpy.linspace(1, 100, 5, endpoint=False))
print(numpy.linspace(1, 100, num=5, endpoint=False))

print(numpy.linspace(1, 100, 5, retstep=True))
print(numpy.linspace(1, 100, num=5, retstep=True))

print(numpy.random.randint(2,100, 15))
print(numpy.random.randint(2,100, size=15))
print(numpy.random.randint([50,20, 23], 100)) # მასივი შდგება სამი შემთხვევითი ელემენტისგან 50-დან 100-მდე , 20--დან 100-მდე და 23--დან 100-მდე
print(numpy.random.randint(2,100, size=[15,2]))
```

ატრიბუტები

```
import numpy

b=numpy.array([[22,3], [4,5]]) –გვაქვს მასივი
print(b)
print(b.shape) ორი ორზე
print(b.size) -4 ელემენტი
print(b.itemsize) ბაიტი თითოეული ინტეჯერის ზომა 4 ბაიტი
print(b.dtype) ამ შემთხვევაში ინტეჯერი არის 4-იანი, თუ უფრო დიდი იქნებოდა ინტეჯერი, მეტი იქნებოდა ზომაც
```

indeqsebi

შევქმენით ნამპაი ერეი და გამოგვყავს ყველა ელემენტი

```
import numpy

b=numpy.array([[220,3], [4,5]])
print(b)
print("--------")
print(b[0: ])
print("--------")
print(b[1: ])
print("--------")
print(b[0:, 1:])  გამოგვყავს ელემენტები ნული ინდექსიდან ბოლომდე და თითოეულიდან პირველიდან ბოლომდე, პირველი არგუმენტი მიუთითებს რიგს, მეორე სვეტს
```

გვაქვს ორგანზომილებიანი მასივი

```
import numpy

b=numpy.array([[1,3,4,5], [6,7,8,9]])
print(b)
print(b[::,::2])
print(b[[0,1], [2,3]]) # აიღე ნულოვანი და ერთი ინდექსის მასივი, ნულოვანიდან აიღე 2 ინდექის ელემენტი, ერთიდან აიღე 3 ინდექსის ელემენტი

print(b[b>4]) # წამოიღე ყველა ელემენტი, რომელიც 4-ზე მეტია

print(b[b>4]*2) # წამოიღე ყველა ელემენტი, რომელიც 4-ზე მეტია და გააამრავლე ორზე

print(b+2) # ყველა ელემენტს მიუმატებს ორს
```

```
c=numpy.array([2,3,4,5,6,7])
d=numpy.array([4,5,3,2,3,4])
print(c+d) # შეკრება შგვიძლია ერთნაირი მასივების

print(numpy.sort(c+d))



a=numpy.random.randint(0, 100, size=[5,4]) #  შემთხვევითი რიცხვებისგან შექმენი მასივი 5 ელემენტისგან, თითოეულში იყოს 4 ელემენტი

print(a)

b=numpy.array([[1,2,3,4], [5,6,7,8]])
c=numpy.array([[4,3,2,1],[8,7,6,5]])
print(b+c)
```

```
import numpy as np
a=np.arange(10)  #  0-დან 10-მდე
e=np.arange(5)
w=np.arange(5)
print(a)
print(e)
print(w)

c=a.reshape(5,2) # შგვიძლია შიპიეს შცვლა, 10 ელემენტი 5/2 ზე
print(c, c.shape) # რატომ გამოვიდა, იმიტომ, რომ იყოფა, ვცადოთ 5 და 3-ის გადაცემა, რაც გამოიწვევს გამონაკლისს
ამისთვის გვაქვს resize()

b=np.reshape(a, (5,2))
print(b, b.shape)
print("--------------------")

d=np.reshape(a, (5,2), order="F") # order="F" აიღებს შუიდან
print(d, d.shape)
print("--------------------")
print(a.reshape(5,2))
b=np.resize(a, (5,3)) - ამატებს ელემენტებს, რაც დააკლდება

print(b, b.shape)
```

```
print("--------------------")
print(b.flatten()) # ბ-სგან ქმნის ერთ განზომილებიან ერეის
print(b.ravel()) # აკეთებს იგივეს, რასაც flatten, მაგრამ flatten უფრო გამოიყენება, რადგან ასლს ქმნის, ხოლო ravel დროს თუ ასლს შვცვლით , შეიცვლება დედანიც
print(b, b.shape)

print(b.transpose()) ცვლის ფორმას საპირისპიროდ

print(b.swapaxes(0, 1)) დაახლოებით იგივეა, რაც transpose ნულოვენი ღერძი არის იქსი და 1 იგრეკი, მათ გაცვალეს ადგილები

print(np.concatenate((a,e, w))) - აერთიანებს
print(e)
print(w)



print('vstack', np.vstack((e,w)))     -  ვერტიკალურად დააწყო ერთმანეთთან
print('hstack', np.hstack((e,w))) ჰორიზონტალურად დააწყო ერთმანეთთან
print('split', np.split(a,5)) - გაყოფს 5 ნაწილად სადაც გამოვა დამოუკიდებელი ერეიები, როცა რესაიზი ცვლიდა საიზს, ამან შექმნა დამოუკიდებელი ერეიები
print('insert', np.insert(a,(1,5), 50 )) ჩასვამს 50 პირველ და მეხუთე პოზიციებზე
print('insert', np.insert(a,(1,5, 3, 8), 1000 )) ჩასვამს რამდენიმე პოზიციაზე
print('append', np.append(a, (10000))) ჩასვამს ბოლოში
new=np.array([[1,6],[7,8]])
print(np.delete(new,2)) წაშლის 2 ინდეექსის ელემენტს და შედეგს აერთიანებს
```

13. შესავალი Django-ში: რა არის Django? Django-ს სამუშაო გარემოს დაყენება; ახალი Django პროექტის შექმნა; პროექტის სტრუქტურის გააზრება MVC/MVT დიზაინის ნიმუშის მიხედვით.
14. Django Admin Interface: Django ადმინისტრატორის საიტის დაყენება.
15. Django ORM – ობიექტების ურთიერთობის შეთავსება: რა არის Django ORM? Django პროექტის დაყენება ORM-ით (Database Setting - Postgres); მოდელების განსაზღვრა – მოდელის სინტაქსი, ველები და ველის პარამეტრები ჯანგოს მოდელებში; მონაცემთა ბაზის მიგრაცია – მიგრაციის გაგება, მიგრაციის შექმნა და გამოყენება, მოდელების შეცვლა და სქემის ცვლილებების მართვა; Querysets - ფილტრები, გამორიცხვები, ანოტაციები, შეკვეთა და მონაცემების გაერთიანება; მოდელის ურთიერთობები: ერთი--ერთთნ, ერთი-მრავალთან და მრავალი-მრავალთან ურთიერთობები, დაკავშირებული მონაცემების შექმნა და მოთხოვნა, კასკადური და on_delete ქცევის გაგება; მოდელის მემკვიდრეობა - აბსტრაქტული საბაზისო კლასები; მოდელის მენეჯერები.
16. Views and URL Patterns: Introduction to views and URL patterns - Understanding the role of views in Django, Creating a simple view and connecting it to a URL, URL-ის წარმოდგენები და

შაბლონები: URL-ის წარმოდგენების და შაბლონების შესავალი - წარმოდგენები როლის გააზრება Django-ში, მარტივი Views-ს შექმნა და URL-მისამართთან დაკავშირება, საბაზისო HTML შინაარსის რენდერირება Views-ში, ფუნქციებზე დაფუძნებული Views, (FBVs) - ფუნქციებზე დაფუძნებული Views-ის შექმნა და გამოყენება, მონაცემების გადაცემა Views-დან შაბლონებში, მომხმარებლის შეყვანის და ფორმების მართვა Views-ში; კლასზე დაფუძნებული Views (CBV) - კლასზე დაფუძნებული Views შესავალი (get, post მეთოდები), CBV-ების უპირატესობების გაგება FBV-ებთან შედარებით, ხშირად გამოყენებული CBV-ების დანერგვა, როგორიცაა ListView და DetailView; URL-ის შაბლონების განსაზღვრა Views-თვის; შაბლონის რენდერი - Django შაბლონების გამოყენება Views-ში, შაბლონის კონტექსტში და მონაცემების გადაცემა შაბლონებში; გადამისამართებები და HTTP პასუხები - მომხმარებლების გადამისამართება სხვადასხვა URL-ებზე, HTTP პასუხების გაგზავნა სხვადასხვა სტატუსის კოდებით; დეკორატორების ნახვა - Django view დეკორატორების გააზრება, დეკორატორების გამოყენება ავთენტიფიკაციისა და ნებართვებისთვის; Error Handling და Exception Views - შეცდომის გვერდების error pages და გამონაკლისების დამუშავება view-ში, გავრცელებული HTTP შეცდომების მართვა (404, 500 და ა.შ.); პაგინაცია - პაგინაციის განხორციელება view-ში მონაცემთა დიდი ნაკრებისთვის Django-ს ჩაშენებული პაგინაციის მხარდაჭერის გამოყენებით; Middleware and View Processing - Django Middleware და მისი როლის გააზრება view-ს დამუშავებაში.
17. ჯანგოს შაბლონები: შესავალი ჯანგოს შაბლონებში; შაბლონის სინტაქსი და ცვლადები; შაბლონის ტეგები და ფილტრები (მნიშვნელოვანი ამ განყოფილებაში).
18. Forms ფორმები და User Input მომხმარებლის შეყვანა: ფორმების აგება Django-ში; ფორმის ვალიდაცია და მომხმარებლის შეყვანის დამუშავება.
19. Static Files and Media Handling: სტატიკური ფაილების მართვა (CSS, JS, images); მომხმარებლის მიერ ატვირთული მედია ფაილების მართვა.
20. Django Authentication: მომხმარებლის ავტორიზაცია და რეგისტრაცია; შესვლისა და გამოსვლის ფუნქციონალი.
21. სიგნალები ჯანგოში: სიგნალები ჯანგოში შესავალი; სიგნალების განსაზღვრა და დაკავშირება კონკრეტულ მოვლენებთან.
22. Messages Framework-შეტყობინებების პლატფორმა: Django-ს შეტყობინებების პლატფორმის გამოყენება მომხმარებლებისთვის შეტყობინებების გამოსახვისთვის; შეტყობინების სხვადასხვა დონე და მათი გამოყენება (წარმატება, გაფრთხილება, შეცდომა და ინფორმაცია).

GUI, PyQt5 library, labels, buttons, Qt Designer, ComboBox
Images, pop-ups, message boxes, menus
Project with GUI and classes
Parallel programming, threads, programming using threads, threading library, time library.
concurrent.futures library, threads in concurrent.futures, processes, programming using processes
Project with threads
Relational databases, sqlite3 library, sql queries with Python, CREATE TABLE, INSERT, SELECT
Secure work with SQLite, working with databases using classes and modules, UPDATE, DELETE
Non-relational databases, MongoDB, insert_one, insert_many, find, delete_one, delete_many, update_one, update_many
Project on databases and GUI
Numpy arrays, creating the arrays, data types in arrays, arithmetical operations, numpy methods

Logical operations on the arrays, work with indexes, methods on the arrays
Introduction to Django: What is Django? Setting up a Django development environment; Creating a new Django project; Understanding the project structure according to The MVC/MVT Design Pattern.
Django Admin Interface: Setting up the Django admin site.
Django ORM – Object Relation Mapping: What is Django ORM? Setting up a Django project with ORM (Database Setting - Postgres); Defining Models – Model syntax, fields and field options in Django models; Database Migrations – Understanding migrations, Creating and applying migrations, modifying models and handling schema changes; Querysets - filters, excludes, annotations, ordering, and aggregating data; Model Relationships - One-to-one, one-to-many, and many-to-many relationships, Creating and querying related data, understanding cascading and on_delete behavior; Model Inheritance - Abstract base classes; Model Managers.
Views and URL Patterns: Introduction to views and URL patterns - Understanding the role of views in Django, Creating a simple view and connecting it to a URL, Rendering basic HTML content in views; Function-Based Views (FBVs) - Creating and using function-based views, Passing data to templates from views, Handling user input and forms in views; Class-Based Views (CBVs) - Introduction to class-based views (get, post methods), Understanding the advantages of CBVs over FBVs, Implementing commonly used CBVs like ListView and DetailView; Defining URL patterns for views; Template Rendering - Using Django templates in views, Template context and passing data to templates; Redirects and HTTP Responses - Redirecting users to different URLs, Sending HTTP responses with different status codes; View Decorators - Understanding Django view decorators, Using decorators for authentication and permissions; Error Handling and Exception Views - Customizing error pages and handling exceptions in views, Handling common HTTP errors (404, 500, etc.); Pagination - Implementing pagination in views for large data sets using Django's built-in pagination support; Middleware and View Processing - Understanding Django middleware and its role in view processing.
Django Templates: Introduction to Django templates; Template syntax and variables; Template tags and filters (important in this section).
Forms and User Input: Building forms with Django; Form validation and handling user input.
Static Files and Media Handling: Managing static files (CSS, JS, images); Handling user-uploaded media files.
Django Authentication: User authentication and registration; Login and logout functionality.
Signals in Django: Introduction to signals in Django; Defining and connecting signals to specific events.
Messages Framework: Using Django's messages framework to display notifications to users; Different message levels and their uses (success, warning, error, and info).