Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение

высшего образования

«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт цифрового развития

Кафедра инфокоммуникаций

**ОТЧЕТ**

**ПО ЛАБОРАТОРНОЙ РАБОТЕ №2.22**

**дисциплины «Анализ данных»**

**Вариант 13**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | Выполнил:  Иващенко Олег Андреевич  2 курс, группа ИВТ-б-о-22-1,  09.03.02 «Информационные и вычислительные машины», направленность (профиль) «Программное обеспечение средств вычислительной техники и автоматизированных систем»  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | |  |
|  | | Руководитель практики:  Воронкин Роман Александрович, доцент кафедры инфокоммуникаций  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  (подпись) | |  |
|  |  | |  | |

Отчет защищен с оценкой \_\_\_\_\_\_\_\_\_\_\_ Дата защиты\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ставрополь, 2024 г.

**Тема**: «Тестирование в Python [unittest]»

**Цель**: Приобрести навыки написания автоматизированных тестов на языке программирования Python версии 3.x.

Порядок выполнения работы

Листинг 1.1 – Пример 1 (модуль calc.py)

|  |
| --- |
| def add(a, b):  return a + b  def sub(a, b):  return a - b  def mul(a, b):  return a \* b  def div(a, b):  return a / b |

Листинг 1.2 – Пример 1 (модуль test\_calc.py)

|  |
| --- |
| import calc  def test\_add():  if calc.add(1, 2) == 3:  print("Test add(a, b) is OK")  else:  print("Test add(a, b) is Fail")  def test\_sub():  if calc.sub(4, 2) == 2:  print("Test sub(a, b) is OK")  else:  print("Test sub(a, b) is Fail")  def test\_mul():  if calc.mul(2, 5) == 10:  print("Test mul(a, b) is OK")  else:  print("Test mul(a, b) is Fail")  def test\_div():  if calc.div(8, 4) == 2:  print("Test div(a, b) is OK")  else:  print("Test div(a, b) is Fail")  test\_add()  test\_sub()  test\_mul()  test\_div() |

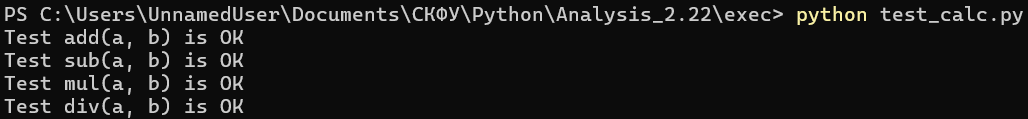


Рисунок 1.1 – Запуск теста

Листинг 1.3 – Пример 2 (unittest)

|  |
| --- |
| import unittest  import calc  class CalcTest(unittest.TestCase):  def test\_add(self):  self.assertEqual(calc.add(1, 2), 3)  def test\_sub(self):  self.assertEqual(calc.sub(4, 2), 2)  def test\_mul(self):  self.assertEqual(calc.mul(2, 5), 10)  def test\_div(self):  self.assertEqual(calc.div(8, 4), 2)  if \_\_name\_\_ == '\_\_main\_\_':  unittest.main() |

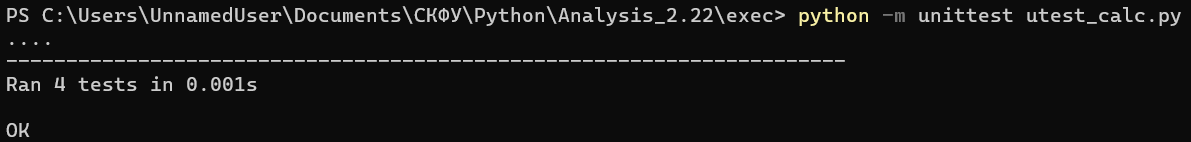


Рисунок 1.2 – Запуск теста unittest

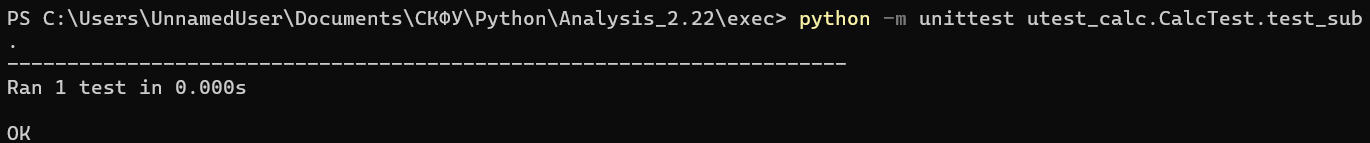


Рисунок 1.3 – Тест отдельного метода модуля (unittest)

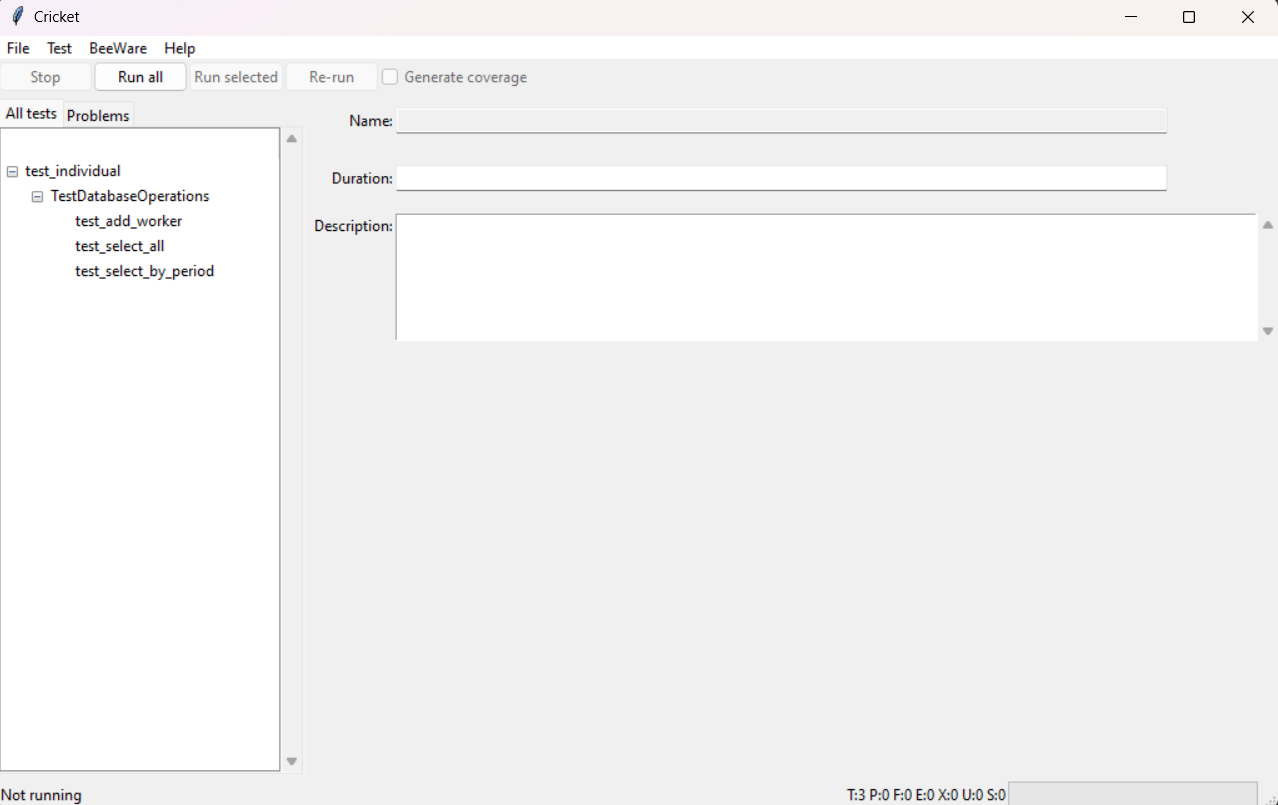


Рисунок 1.4 – Запуск Cricket

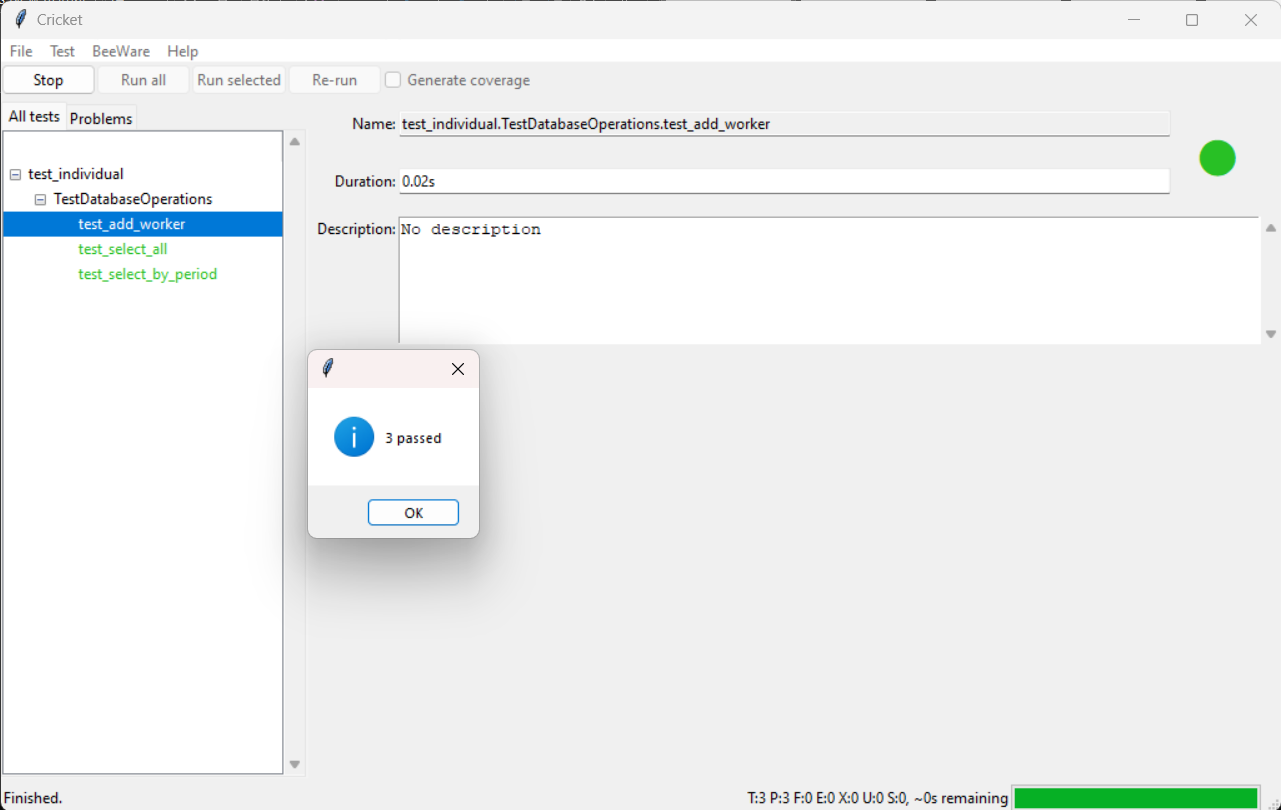


Рисунок 1.5 – Тестирование с помощью Cricket

Индивидуальное задание. Для индивидуального задания лабораторной работы 2.21 добавьте тесты с использованием модуля unittest, проверяющие операции по работе с базой данных.

Листинг 2.1 – Код individual.py

|  |
| --- |
| #!/usr/bin/env python3  # -\*- coding: utf-8 -\*-  """  Для своего варианта лабораторной работы 2.17 необходимо реализовать  хранение данных в базе данных SQLite3. Информация в базе данных  должна храниться не менее чем в двух таблицах.  """  import argparse  from pathlib import Path  import sqlite3  import typing as t  def print\_help():  """  Функция вывода доступных пользователю команд  """  print("list - вывод всех добавленных записей")  print("add - добавление новых записей")  print("find - найти запись по фамилии")  print("exit - завершение работы программы")  def add\_worker(database\_path: Path, name: str, phone: str, year: int) -> None:  """  Функция добавления новой записи, возвращает запись  """  connection = sqlite3.connect(database\_path)  cursor = connection.cursor()  cursor.execute(  """  INSERT INTO workers (worker\_name, phone\_number, worker\_year)  VALUES (?, ?, ?)  """,  (name, phone, year)  )  connection.commit()  connection.close()  def print\_list(staff: t.List[t.Dict[str, t.Any]]) -> None:  """  Функция выводит на экран список всех существующих записей  """  if staff:  line = '+-{}-+-{}-+-{}-+-{}-+'.format(  '-' \* 4,  '-' \* 30,  '-' \* 20,  '-' \* 8  )  print(line)  print(  '| {:^4} | {:^30} | {:^20} | {:^8} |'.format(  "№",  "Ф.И.О.",  "Номер телефона",  "Год"  )  )  print(line)  for idx, worker in enumerate(staff, 1):  print(  '| {:>4} | {:<30} | {:<20} | {:>8} |'.format(  idx,  worker.get('name', ''),  worker.get('phone\_number', ''),  worker.get('year', 0)  )  )  print(line)  else:  print("Список работников пуст.")  def create\_db(database\_path: Path) -> None:  """  Создание базы данных  """  connection = sqlite3.connect(database\_path)  cursor = connection.cursor()  cursor.execute(  """  CREATE TABLE IF NOT EXISTS workers (  worker\_id INTEGER PRIMARY KEY AUTOINCREMENT,  worker\_name TEXT NOT NULL,  phone\_number TEXT NOT NULL,  worker\_year INTEGER NOT NULL  )  """  )  connection.close()  def select\_all(database\_path: Path) -> t.List[t.Dict[str, t.Any]]:  """  Выбор всех записей из базы данных  """  connection = sqlite3.connect(database\_path)  cursor = connection.cursor()  cursor.execute(  """  SELECT workers.worker\_name, workers.phone\_number, workers.worker\_year  FROM workers  """  )  rows = cursor.fetchall()  connection.close()  return [  {  "name": row[0],  "phone\_number": row[1],  "year": row[2],  }  for row in rows  ]  def select\_by\_period(  database\_path: Path, period: int  ) -> t.List[t.Dict[str, t.Any]]:  """  Выборка по периоду  """  connection = sqlite3.connect(database\_path)  cursor = connection.cursor()  cursor.execute(  """  SELECT workers.worker\_name, workers.phone\_number, workers.worker\_year  FROM workers  WHERE (strftime('%Y', date('now')) - workers.worker\_year) >= ?  """,  (period,)  )  rows = cursor.fetchall()  connection.close()  return [  {  "name": row[0],  "phone\_number": row[1],  "year": row[2],  }  for row in rows  ]  def main(command\_line=None):  file\_parser = argparse.ArgumentParser(add\_help=False)  file\_parser.add\_argument(  "--db",  action="store",  required=False,  default=str(Path.home() / "workers.db"),  help="Название файла базы даанных"  )  parser = argparse.ArgumentParser("workers")  parser.add\_argument(  "--version",  action="version",  version="%(prog)s 0.1.0"  )  subparsers = parser.add\_subparsers(dest="command")  add = subparsers.add\_parser(  "add",  parents=[file\_parser],  help="Add a new worker"  )  add.add\_argument(  "-n",  "--name",  action="store",  required=True,  help="Имя работника"  )  add.add\_argument(  "-p",  "--phone",  action="store",  help="Номер телефона работника"  )  add.add\_argument(  "-y",  "--year",  action="store",  required=True,  help="Дата нанятия"  )  \_ = subparsers.add\_parser(  "display",  parents=[file\_parser],  help="Вывести на экран всех работников"  )  select = subparsers.add\_parser(  "select",  parents=[file\_parser],  help="Выборка работников"  )  select.add\_argument(  "-p",  "--period",  action="store",  type=int,  required=True,  help="Требуемый период"  )  args = parser.parse\_args(command\_line)  db\_path = Path(args.db)  create\_db(db\_path)  if args.command == "add":  add\_worker(db\_path, args.name, args.phone, args.year)  elif args.command == "display":  print\_list(select\_all(db\_path))  elif args.command == "select":  print\_list(select\_by\_period(db\_path, args.period))  pass  if \_\_name\_\_ == "\_\_main\_\_":  """  Основная программа  """  main() |

Листинг 2.2 – Код test\_individual.py

|  |
| --- |
| import unittest  from pathlib import Path  import tempfile  import os  from individual import create\_db, add\_worker, select\_all, select\_by\_period  «»»  Для индивидуального задания лабораторной работы 2.21 добавьте тесты с  использованием модуля unittest, проверяющие операции по работе с базой данных.  “””  class TestDatabaseOperations(unittest.TestCase):  def eard(self):  self.db\_fd, self.db\_path = tempfile.mkstemp()  create\_db(Path(self.db\_path))  def eardown(self):  os.close(self.db\_fd)  os.unlink(self.db\_path)  def test\_add\_worker(self):  add\_worker(Path(self.db\_path), “Петров Петр”, “79998886452”, 2000)  workers = select\_all(Path(self.db\_path))  self.assertEqual(len(workers), 1)  self.assertEqual(workers[0][‘name’], “Петров Петр”)  self.assertEqual(workers[0][‘phone\_number’], “79998886452”)  self.assertEqual(workers[0][‘year’], 2000)  def test\_select\_all(self):  add\_worker(Path(self.db\_path), “Петров Петр”, “79998886452”, 2000)  add\_worker(Path(self.db\_path), “Иванов Иван”, “79995554291”, 2015)  workers = select\_all(Path(self.db\_path))  self.assertEqual(len(workers), 2)  def test\_select\_by\_period(self):  add\_worker(Path(self.db\_path), “Петров Петр”, “79998886452”, 2000)  add\_worker(Path(self.db\_path), “Иванов Иван”, “79995554291”, 2015)  workers = select\_by\_period(Path(self.db\_path), 10)  self.assertEqual(len(workers), 1)  self.assertEqual(workers[0][‘name’], “Петров Петр”)  if \_\_name\_\_ == ‘\_\_main\_\_’:  unittest.main() |

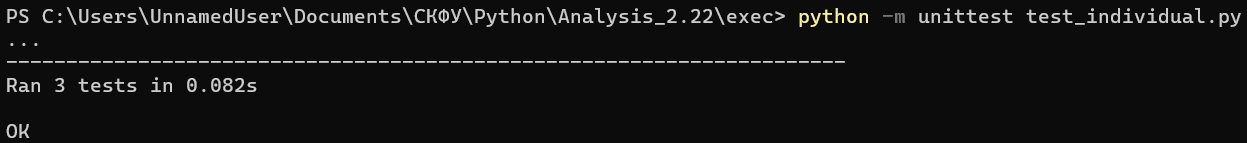


Рисунок 2.1 – Тестирование individual.py (unittest)

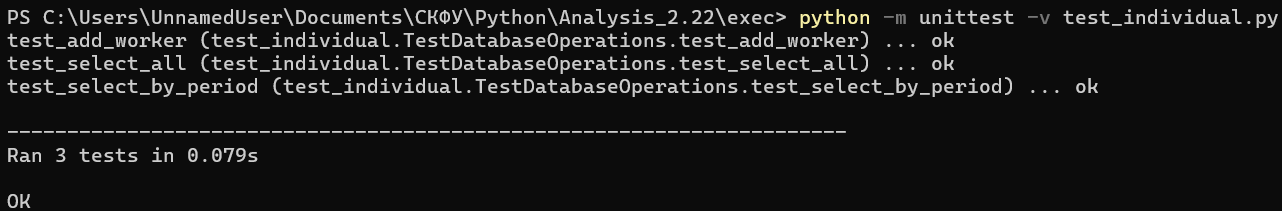


Рисунок 2.2 – Подробное тестирование individual.py (unittest)

Контрольные вопросы

1. Для чего используется автономное тестирование?

Автономное тестирование используется для проверки отдельных частей программы, таких как функции и классы, изолированно от остальной системы, чтобы гарантировать их правильную работу.

1. Какие фреймворки Python получили наибольшее распространение для решения задач автономного тестирования?

Наиболее распространённые фреймворки для автономного тестирования в Python – unittest, pytest и nose.

1. Какие существуют основные структурные единицы модуля unittest?

Основные структурные единицы модуля unittest:

* TestCase – класс для создания тестов;
* TestSuite – класс для объединения нескольких тестов;
* TestLoader – класс для загрузки классов;
* TextTestRunner – класс для запуска тестов;
* TestResult – класс для хранения результатов тестов.

1. Какие существуют способы запуска тестов unittest?

Unittest можно запустить из командной строки с помощью команды python -m unittest <названи\_файла> или с помощью программного вызова через unittest.main().

1. Каково назначение класса TestCase?

Класс TestCase используется для создания тестов, определяя методы, начинающиеся с test, которые содержат проверяемые условия.

1. Какие методы класса TestCase выполняются при запуске и завершении работы тестов?

Методы класса TestCase, выполняемые при запуске и завершении работы тестов – setup() (выполняется перед каждым тестом) и tearDown() (выполняется после каждого теста).

1. Какие методы класса TestCase используются для проверки условий и генерации ошибок?

Методы класса TestCase для проверки условий и генерации ошибок:

* assertEqual();
* assertTrue();
* assertFalse();
* assertRaises();
* assertIn();
* assertNotIn();
* assertIs();
* assertIsNot();
* assertIsNone();
* assertIsNotNone().

1. Какие методы класса TestCase позволяют собирать информацию о самом тесте?

Методы класса TestCase для собора информации о тесте:

* id() – возвращает уникальный идентификатор теста;
* shortDescription() – возвращает краткое описание теста (если задано в виде строки документации).

1. Каково назначение класса TestSuite? Как осуществляется загрузка тестов?

* Класс TestSuite используется для объединения нескольких тестов в одну группу для их совместного запуска. Загрузка тестов осуществляется с помощью методов addTest(), addTests() и с помощью класса TestLoader.

1. Каково назначение класса TestResult?

Класс TestResult используется для хранения и отображения результатов тестирования, включая информацию о пройденных, проваленных и пропущенных тестах.

1. Для чего может понадобиться пропуск отдельных тестов?

Пропуск отдельных тестов может понадобиться, если тест ещё не реализован, зависит от внешних условий или временно неактуален.

1. Как выполняется безусловный и условных пропуск тестов? Как выполнить пропуск класса тестов?

Безусловный пропуск тестов можно выполнить с помощью декоратора @unittest.skip(«причина»).

Условный пропуск тестов можно выполнить с помощью декораторов @unittest.skipIf(condition, «причина») и @unittest.skipUnless(condition, «причина»).

Пропуск класса тестов можно выполнить с помощью декорирования всего класса с помощью @unittest.skip(«причина»).

**Выводы**: В процессе выполнения лабораторной работы были изучены способы тестирования программ/модулей с помощью различных инструментов, начиная от встроенных в Python и заканчивая дополнительными программными обеспечениями для проведения тестирования модуля с помощью GUI. Были проработаны примеры лабораторной работы, а также выполнено индивидуальное задание.