абочая тетрадь № 8

|  |
| --- |
| Основной задачей компьютера является обработка данных. Под обработкой данных понимается как вычислительные процедуры, так и процедуры, преобразующие одни данные в другие данные.  Данные – это форма представление информации, доступная для обработки. Формат представления данных – это последовательность бит (байт). Физически данные хранятся в виде файлов или потоков данных на физических носителях информации. Поток данных – это абстракция для доступа к данным из файлов, периферийных устройств и т.д.  Список – структура данных, состоящая из узлов, хранящих как данные, так и ссылку на следующий элемент. |

|  |
| --- |
| **1. Теоретический материал** |
| Зачастую возникает необходимость работы не с отдельными переменными, а с целым набором данных, как с единым целым. При таком подходе целесообразно использовать структуры данных – контейнеры, информация в которых храниться характерным образом. В зависимости от конкретной задачи, данные необходимо хранить в подходящем формате. Для этого в программировании существует набор различных структур данных, обеспечивающих нас различными форматами. Наиболее простой и распространенной структурой данных выступает массив. Многие другие структуры данных производны от массивов.  ***Массив*** – набор однотипных элементов, доступ к которым осуществляется по индексу. Элементы массива расположены в памяти последовательно друг за другом.    Рисунок 1 – Схема массива  Список – структура данных, состоящая из узлов, в которых хранятся данные, а так же ссылку на следующий (и/или предыдущий) элемент.    Рисунок 2 – Схема динамического списка  В некоторых задачах необходимо использовать структуру данных очередь, которая является частным случаем списка. В данном случае мы сохраняем и имеем доступ к элементам не по индексу, а по принципу «первый вошел, первый вышел» FIFO – «*first in, first out*».    Рисунок 3 – Схема очереди  Еще одной ключевой структурой данных выступает стек, который функционирует по принципу: «последний вошел, первый вышел» LIFO – «*last in, first out*».    Рисунок 4 – Схема стека |

|  |  |
| --- | --- |
| **2. Пример** | |
| ***Задача:*** | |
|  | Дана строка. Используя только структуру данных «стек» и операции над стеком необходимо получить строку «в обратном порядке» и вывести её на экран. В Python для эмуляции структуры данных «стек» можно использовать списки. |
| ***Решение:*** | |
|  | С использованием стека задача решается следующим образом. Каждый символ заданной величины, начиная с первого, размещаем в стеке. В результате на вершине стека окажется последний символ, “под” ним — предпоследний и т.д. Извлекая символы из стека, выводим их на экран в порядке, обратном исходному.  Пример возможного решения:  word = 'экзистенциализм'  stack = list(word)  while len(stack)>0:  ch = stack.pop()  print(ch, end='') |

|  |  |  |
| --- | --- | --- |
| **3. Задания** | | |
| 1. | ***Задача:*** | |
|  | Дана величина **a** строкового типа из четного количества символов. Получить и напечатать величину **b**, состоящую из символов первой половины величины **a**, записанных в обратном порядке, после которых идут символы второй половины величины **a**, также записанные в обратном порядке. Например, при **а** = «привет» **b** должно быть равно «ирптев». |
| ***Решение:*** | |
|  | a = input()  g = int(len(a) / 2)  b = a[g - 1::-1]  b = b + a[-1:g - 1:-1]  print(b) |
| 2. | ***Задача\*:*** | |
|  | Написать программу, которая определяет, является ли введенная скобочная структура правильной. Примеры правильных скобочных выражений: (), (())(), ()(), ((())), неправильных — )(, ())((), (, )))), ((()). |
| ***Решение:*** | |
|  | def is\_correct\_brackets(text):  while '()' in text or '[]' in text or '{}' in text:  text = text.replace('()', '')  text = text.replace('[]', '')  text = text.replace('{}', '') |

|  |  |  |
| --- | --- | --- |
| **Тест 8** | | |
| **1.** | Выберете этап, который не входит в этапы создания программы: | |
|  | А) проектирование Б) бенчмаркинг  В) написание кода Г) тестирование и отладка |
| ***Ответ:*** | |
|  |  |
| **2.** | Какие парадигмы программирования существуют (множественный выбор) | |
|  | А) структурное программирование  Б) объектно-ориентированное  В) функциональное программирование  Г) все ответы верны |
| ***Ответ:*** | |
|  |  |
| **3.** | Что НЕ относится к принципам структурного программирования | |
|  | А) Программа представляется в виде иерархической структуры блоков  Б) Разработка программы ведется методом «сверху вниз»  В) Разработка программы ведется методом «снизу вверх»  Г) Тело программы содержит управляющие структуры: последовательность, ветвление, цикл и подпрограммы |
| ***Ответ:*** | |
|  |  |
| **4.** | Линейный список с доступом только к последнему (верхнему) элементу называется | |
|  | А) стеком Б) очередью  В) двухсторонней очередью Г) кольцом |
| ***Ответ:*** | |
|  |  |

|  |  |  |
| --- | --- | --- |
| **5.** | Контейнер, работа в котором производиться по принципу FIFO (первый пришел - первый ушел) это – | |
|  | А) Стек Б) Двухсторонняя очередь  В) Очередь Г) Список |
| ***Ответ:*** | |
|  |  |
| **6.** | В чём особенности стека ? | |
|  | А) открыт с обеих сторон на вставку и удаление  Б) доступен любой элемент  В) открыт с одной стороны на вставку и удаление  Г) закрыт с обеих сторон на вставку и удаление |
| ***Ответ:*** | |
|  |  |
| **7.** | Дана следующая последовательность стековых операций. Что будет на вершине стека, когда последовательность завершится?  m = Stack()  m.push('x')  m.push('y')  m.pop()  m.push('z')  m.peek() | |
|  | А) 'x' Б) 'y' В) 'z' Г) стек пуст |
| ***Ответ:*** | |
|  |  |
| **8.** | Из всех приведенных ниже утверждений, касающихся объектов и классов, отметьте правильные. | |
|  | 1) значения полей описывают состояние объекта  2) методы класса описывают поведение объектов  3) поля класса могут быть разного типа  4) поля класса должны быть разного типа  5) в любом классе должно быть как минимум одно поле данных |
| ***Ответ:*** | |
|  |  |

|  |  |  |
| --- | --- | --- |
| **9.** | Термин "наследование" обозначает, что... | |
|  | А) В производных классах присутствует часть состояния родительского класса  Б) Производные классы содержат поля и методы родительского  В) Производные классы наследуют модификаторы доступа членов родительского класса |
| ***Ответ:*** | |
|  |  |
| **10.** | Инкапсуляция – это: | |
|  | А) Свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.  Б) Сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса или копирования прототипа (например, после запуска результатов компиляции и связывания исходного кода на выполнение)  В) Свойство системы, позволяющее объединить данные и методы, работающие с ними в классе, и скрыть детали реализации от пользователя. |
| ***Ответ:*** | |
|  |  |

Реализация задач на языке программирования Python

|  |
| --- |
| Ранее нами было рассмотрено традиционное программирование на Python, когда программа состояла из отдельных модулей, содержащих функции. Такой подход к программированию характерен для структурной парадигмы программирования (или императивной парадигмы) [5]. Структурное программирование большой шаг вперед в построении программ. Однако еще большим шагом выступает объектно-ориентированная парадигма. В названном подходе программа включает отдельные абстрактные типы данных (классы), которые объединяют в себе переменные (поля или свойствами класса), и функции (методы класса). |

|  |
| --- |
| **1. Теоретический материал** |
| Объектно-ориентированная парадигма включает в себя три ключевых принципа [5]:   * Инкапсуляция; * Полиморфизм; * Наследование.   Рассмотрим эти понятия на практических примерах. Инкапсуляция – это объединение в одном объекте программного кода и данных таким образом, что внутренняя часть объекта может быть скрыта от пользователя для внешней работы.  Приведем пример инкапсуляции в Python. Ниже определен класс (новый тип данных) и создан объект, называемый экземпляром класса. Имена классов в Python рекомендовано начинать с заглавной буквы "T", подчеркивая тем самым, что речь идет собственном о типе данных.  class TAnimal:  name = ""  def \_\_init\_\_(self, name):  self.name = name  def say(self):  print(self.name)  Теперь создадим экземпляр этого класса. Экземпляр класса представляет собой переменную, с которой можно работать обычным образом.  Animal = TAnimal("Обезьяна")  Animal.say()  Создание класса начинается с ключевого слова **class**. Далее в блоке определяются поля (переменные) и методы (функции) класса. Методы определяются, как обычные функции и могут возвращать значения. Ключевое отличие заключается в том, что у всех методов есть обязательный первый параметр, который традиционно называется словом **self.** В данный параметр передается ссылка на экземпляр класса. Когда внутри класса в методе требуется обратиться к своему полю, необходимо задействовать конструкцию **self.имя\_поля**. Заметим, что при вызове методов класса первый параметр не задается.  У классов присутствует специальная конструкция, с именем **\_\_init\_\_**, которая называется конструктором класса. Данная конструкция вызывается в момент создания экземпляра класса **Animal = TAnimal("Обезьяна")**. Конструкторы классов могут при этом иметь любое заданное количество параметров.  Допустим теперь требуется создать абстрактный тип данных для описания конкретного животного, например кошки. В данном случае целесообразно использовать еще один принцип объектно-ориентированного программирования – наследование классов. Данный принцип заключается в возможности определения новых классов, на базе уже существующих. При этом новый класс (класс-наследник) будет иметь все поля и методы наследуемого класса (родительский класс или суперкласс). Ниже представлен пример реализации наследования на языке Python.  class TAnimal:  name = ""  def \_\_init\_\_(self, name):  self.name = name  def say(self):  print(self.name)  class TCat(TAnimal):  def may(self):  print("Мяу!")  Cat = TCat("Кошка")  Cat.say()  Cat.may()  В примере показано, что у класса-наследника сохранился конструктор базового класса и метод **say**. Наследный класс, также как и исходный имеет конструктор, который принимает в качестве параметра название животного, что в данном случае может оказаться излишним. Для разрешения данной проблемы мы воспользуемся еще одним принципом объектно-ориентированного программирования – механизмом полиморфизма. Полиморфизм – это возможность замены методов у классов наследников при наследовании. Приведем пример кода, в котором нет необходимости передавать в конструкторе название класса "Кошка".  class TCat(TAnimal):  def \_\_init\_\_(self):  super().\_\_init\_\_("Кошка")  def may(self):  print("Мяу!")  Cat = TCat()  Cat.say()  Cat.may()  Результат работы данного кода будет идентичным, однако теперь при использовании класса-наследника нет необходимости передавать параметры в конструкторе. Полиморфное перекрытие методов достигается путем простого объявлением метода в классе-наследнике (в данном случае конструктора). Если в результате написания кода метода возникает необходимость вызвать метод родительского класса, то для этого можно использовать функцию **super()**, которая просто возвращает ссылку на родительский класс. Рассмотрим еще один пример полиморфного поведения. Допустим есть класс:  class TDo:  def Operation(self, x, y):  return x + y  def Run(self):  x = int(input("Enter x > "))  y = int(input("Enter y > "))  z = self.Operation(x, y)  print("Result = " + z.\_\_str\_\_())  Do = TDo()  Do.Run()  Используя полиморфизм можно заменить реализацию метода **Operation** на другую в классе-наследнике:  class TDo2(TDo):  def Operation(self, x, y):  return x \* y |

|  |  |
| --- | --- |
| **2. Пример** | |
| ***Задача:*** | |
|  | Разработать абстрактную виртуальную модель процесса обучения. В программе должны быть классы: ученики, учитель, кладезь знаний. |
| ***Решение (код программы):*** | |
|  | Для реализации виртуальной модели обучения нам потребуется три класса – "учитель", "ученик", "данные". Классы учителя и ученика во многом схожи, следовательно, их они могут наследоваться от родительского класса "человек". Однако в контексте данной задачи у учителя и ученика вряд ли найдутся общие атрибуты.  Определим, что должны уметь объекты для решения задачи "увеличить знания":   * Ученик должен уметь брать информацию и превращать ее в свои знания. * Учитель должен уметь учить группу учеников. * Данные могут представлять собой список знаний. Элементы будут извлекаться по индексу.   class Data:  def \_\_init\_\_(self, \*info):  self.info = list(info)  def \_\_getitem\_\_(self, i):  return self.info[i]  class Teacher:  def teach(self, info, \*pupil):  for i in pupil:  i.take(info)  class Pupil:  def \_\_init\_\_(self):  self.knowledge = []  def take(self, info):  self.knowledge.append(info)  lesson = Data('class', 'object', 'inheritance', 'polymorphism', 'encapsulation')  marIvanna = Teacher()  vasy = Pupil()  pety = Pupil()  marIvanna.teach(lesson[2], vasy, pety)  marIvanna.teach(lesson[0], pety)  print(vasy.knowledge)  print(pety.knowledge) |

|  |  |  |
| --- | --- | --- |
| **3. Задания** | | |
| 1. | ***Задача:*** | |
|  | Напишите программу со следующим функционалом. В программе должен присутствовать класс "Воин". От него создаются два экземпляра-юнита с индикатором здоровья в 100 единиц. В случайном порядке юниты атакуют друг друга, при этом атакующий экземпляр здоровья не теряет, а здоровье атакуемого уменьшается на 20 единиц после каждого удара. После каждой атаки необходимо выводить сообщение, какой юнит атаковал, и сколько у противника осталось здоровья. Как только у кого-то заканчивается ресурс здоровья, программа завершается сообщением о том, кто одержал победу. |
|  | ***Решение (код программы):*** | |
|  | import random  class Warrior:  def \_\_init\_\_(self, name, health=100):  self.name = name  self.health = health  def attack(self, enemy):  damage = 20  print(f"{self.name} атакует {enemy.name}! {enemy.name} теряет {damage} единиц здоровья.")  enemy.health -= damage  def battle(warrior1, warrior2):  while warrior1.health > 0 and warrior2.health > 0:  attacker = random.choice([warrior1, warrior2])  defender = warrior2 if attacker == warrior1 else warrior1  attacker.attack(defender)  if warrior1.health <= 0:  print(f"{warrior1.name} побежден! Победитель: {warrior2.name}")  else:  print(f"{warrior2.name} побежден! Победитель: {warrior1.name}")  warrior1 = Warrior("Воин 1")  warrior2 = Warrior("Воин 2")  print(f"Начинается бой между {warrior1.name} и {warrior2.name}!")  battle(warrior1, warrior2) |
|  |  |  |