**Домашняя работа 4**

**Задание 1:**

Реализовать свою мини-версию Make.

Для описания задач, зависимостей и команд можно использовать язык реализации (если это скриптовый язык в духе Питона) или же реализовать собственный язык с помощью средств из ДЗ 3. Если у вас нет хороших идей на этот счет, то используем синтаксис make, но позволяем отступы делать "по-человечески", а не только с помощью "\t".

Для определения порядка выполнения задач должен использоваться алгоритм топологической сортировки. Уже выполненные задачи повторно не выполняются.

Для определения изменений в зависимостях задач (файлах и других видах данных) необходимо использовать не время изменения файла, а хеш-значения, см. https://medium.com/@buckaroo.pm/build-systems-should-use-hashes-over-timestamps-54d09f6f2c4

Обратите внимание, вам нужно хешировать содержимое файлов. То есть для хранения аналога файловой системы с информацией о файлах вполне можно использовать тот же JSON-объект, а вот хешировать сами данные лучше чем-нибудь проверенным, с малой вероятностью коллизий. Посмотрите стандартную библиотеку. Для Питона это https://docs.python.org/3/library/hashlib.html

Проверить работоспособность реализованной системы автоматизации сборки на примере civgraph (том варианте, где создаются файлы!), а также на собственном примере компиляции приложения.

Иными словами, идеальное решение подразумевает настоящую, пусть и примитивную, систему сборки, которая работала бы с реальными файлами и произвольными командами, а также включала в себя БД в каком-то виде для хранения состояния между запусками. Обязательно приведите примеры работы вашей утилиты.

|  |
| --- |
| import argparse  from argparse import ArgumentParser  import hashlib  import os  import json  class SmartFormatter(argparse.HelpFormatter):  def \_split\_lines(self, text, width):  if text.startswith('R|'):  return text[2:].splitlines()  return argparse.HelpFormatter.\_split\_lines(self, text, width)  class minimake:  def \_\_init\_\_(self):  # Инициализация базовых структур данных  self.tasks = {} # Словарь задач (имя -> информация о задаче)  self.task\_status = {} # Словарь статусов задач (имя -> выполнена или нет)  self.file\_hashes = {} # Словарь хэшей файлов (имя файла -> хэш)  def taska(self, name, depends=None):  def decorator(func):  # Добавление информации о задаче в словарь  self.tasks[name] = {  'func': func, # Сохранение функции задачи  'depends': depends if depends else [], # Сохранение зависимостей (если есть)  }  self.task\_status[name] = False # Инициализация статуса задачи как невыполненной  return func  return decorator  def hash\_file(self, filename):  # Вычисление хэша файла с использованием SHA-256  hasher = hashlib.sha256()  with open(filename, 'rb') as file:  while True:  data = file.read(4096)  if not data:  break  hasher.update(data)  return hasher.hexdigest()  def execute\_task(self, task\_name):  if not self.task\_status[task\_name]:  task\_info = self.tasks[task\_name]  dependencies = task\_info['depends']  for dependency in dependencies:  if dependency not in self.file\_hashes:  # Если зависимость не была ранее обработана, вычисляем её хэш  if os.path.exists(dependency): # Проверка существования файла  self.file\_hashes[dependency] = self.hash\_file(dependency)  else:  print(f"Зависимость '{dependency}' не найдена.")  return  current\_hash = self.hash\_file(dependency)  if current\_hash != self.file\_hashes[dependency]:  print(f"Задача '{task\_name}' устарела. Пересчет...")  break  else:  print(f"Выполнение задачи: {task\_name}")  task\_info['func']()  self.task\_status[task\_name] = True  def build(self, target):  execution\_order = []  def dfs(node):  # Рекурсивный обход в глубину для определения порядка выполнения задач  for dependency in self.tasks[node]['depends']:  if not self.task\_status[dependency]:  dfs(dependency)  execution\_order.append(node)  dfs(target)  for task\_name in execution\_order:  self.execute\_task(task\_name)  if \_\_name\_\_ == "\_\_main\_\_":  make = minimake()  @make.taska("input.txt")  def create():  print("Создание input.txt...")  with open("civgraph.txt","r") as infile:  data = infile.read()  with open("input.txt", "w") as f:  f.write(data)  @make.taska("result.txt", depends=["input.txt"])  def process():  print("Обработка input.txt...")  with open("input.txt", "r") as infile:  data = infile.read()  processed\_data = data.upper()  with open("result.txt", "w") as outfile:  outfile.write(processed\_data)  @make.taska("result\_backup.txt", depends=["result.txt"])  def backup():  print("Создание резервной копии result.txt...")  with open("result.txt", "r") as f:  data = f.read()  with open("result\_backup.txt", "w") as f:  f.write(data)  @make.taska("modify", depends=["result.txt"])  def modify():  print("Изменение файла result.txt...")  with open("result.txt", "r") as f:  data = f.read()  modified\_data = data + "\nIKBO-12-22\ndfsdjfjsdfjsjf\n123\n567"  with open("result.txt", "w") as f:  f.write(modified\_data)  @make.taska("clean")  def clean():  print("Очистка...")  files\_to\_remove = ["result.txt", "input.txt", "result\_backup.txt"]  for file\_name in files\_to\_remove:  if os.path.exists(file\_name):  os.remove(file\_name)  else:  print(f"Файл '{file\_name}' не найден.")  @make.taska("restore", depends=["result\_backup.txt"])  def restore\_from\_backup():  print("Восстановление result.txt из резервной копии...")  with open("result\_backup.txt", "r") as f:  data = f.read()  with open("result.txt", "w") as f:  f.write(data)  parser = ArgumentParser(description='Справка', formatter\_class=SmartFormatter)  parser.add\_argument('command', default='create\_input',  help="R| Option \tDescription\n"  " create - создать входной файл\n"  " process - создать выходной файл\n"  " backup - создать бэкап выходного файла\n"  " modify - изменить выходной файл\n"  " clean - удалить входной и выходной файлы\n"  " restore - восстановить выходной файл из бэкапа\n")  args = parser.parse\_args()  parser.parse\_args()  command = args.command  if command == "create":  make.build("input.txt")  elif command == "process":  make.build("result.txt")  elif command == "backup":  make.build("result\_backup.txt")  elif command == "modify":  make.build("modify")  elif command == "run":  make.build("run")  elif command == "clean":  make.build("clean")  elif command == "restore":  make.build("restore")  else:  print(f"Неизвестная команда: {command}") |



