

Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«СЕВЕРО-КАВКАЗСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Институт перспективной инженерии
Департамент цифровых, робототехнических систем и электроники

ОТЧЕТ
ПО ЛАБОРАТОРНОЙ РАБОТЕ №4.2
дисциплины «Объектно-ориентированное программирование»

Выполнил:
Магомедов Имран Борисович
3 курс, группа «Программная инженерия»,
направленность (профиль) «Разработка
и сопровождение программного
обеспечения», очная форма обучения

(подпись)

Руководитель практики:
Воронкин Р.А., доцент департамента
цифровых, робототехнических систем и
электроники

(подпись)

Отчет защищен с оценкой _____ Дата защиты _____

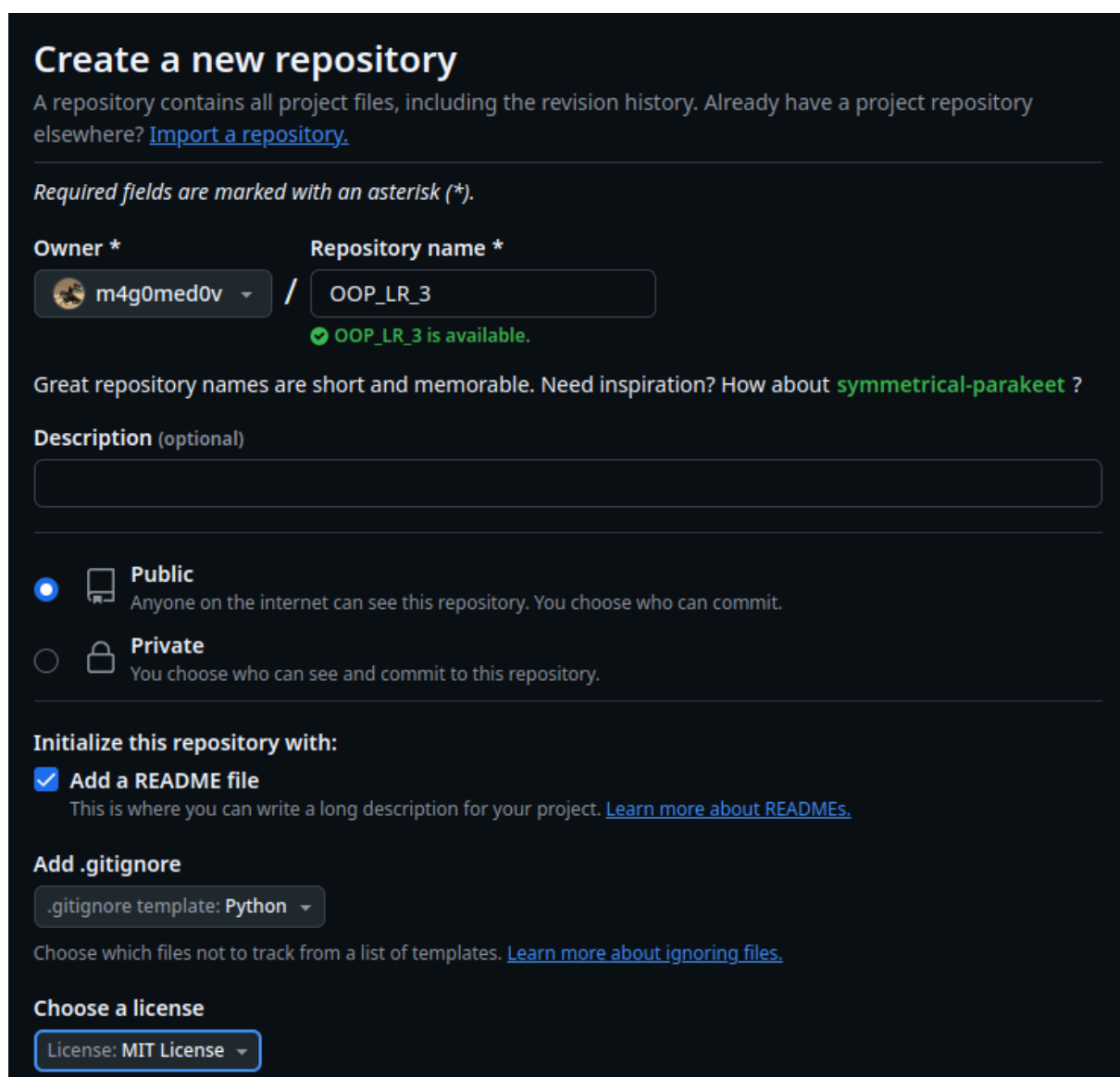
Ставрополь, 2024 г.

Тема: Перегрузка операторов в языке Python

Цель работы: приобретение навыков по перегрузке операторов при написании программ с помощью языка программирования Python версии 3.x.

Методика выполнения работы


1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)


Required fields are marked with an asterisk ().*


Owner *  m4g0med0v / **Repository name ***

✔ OOP_LR_3 is available.

Great repository names are short and memorable. Need inspiration? How about [symmetrical-parakeet](#) ?

Description (optional)

☒  **Public**
Anyone on the internet can see this repository. You choose who can commit.

☐  **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

☒ **Add a README file**
This is where you can write a long description for your project. [Learn more about READMEs.](#)

Add .gitignore

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

3. Выполните клонирование созданного репозитория.

```

~/work git clone https://github.com/m4g0med0v/00P_LR_3.git
Клонирование в «00P_LR_3»...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Получение объектов: 100% (5/5), готово.

```

4. Организуйте свой репозиторий в соответствии с моделью ветвления git-flow.

```

~/work/00P_LR_3 main git switch develop
Переключились на ветку «develop»

```

5. Проработайте примеры лабораторной работы.

```

> .venv 5 class Rational:
150     return False
151
152     def __lt__(self, rhs):
153         if isinstance(rhs, Rational):
154             return self.__float__() < rhs.__float__()
155         else:
156             return False
157
158     def __ge__(self, rhs):
159         if isinstance(rhs, Rational):
160             return not self.__lt__(rhs)
161         else:
162             return False
163
164     def __le__(self, rhs):
165         if isinstance(rhs, Rational):
166             return not self.__gt__(rhs)
167         else:
168             return False
169
170
171 if __name__ == "__main__":
172     r1 = Rational(3, 4)
173     print(f"r1 = {r1}")
174
175     r2 = Rational(5, 6)

```

6. Выполните индивидуальные задания. Приведите в отчете скриншоты работы программ решения индивидуального задания.

Индивидуальное задание №1. Выполнить индивидуальное задание 1 лабораторной работы 4.1, максимально задействовав имеющиеся в Python средства перегрузки операторов.

```
.venv
└─ examples
   ├── example_1.py
   ├── example_2.py
   └── example_3.py
└─ src
   ├── individual_task_1....
   ├── individual_task_2....
   ├── .gitignore
   ├── .pre-commit-config...
   ├── LICENSE
   ├── README.md
   └── requirements.txt

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  # Выполнить индивидуальное задание 1 лабораторной работы 4.1,
5  # максимально задействовав имеющиеся в Python средства перегрузки
6  # операторов.
7
8
9  class Pair:
10     def __init__(self, firts: int, second: int) -> None:
11         # Проверка корректности диапазона
12         if not (isinstance(firts, int) and isinstance(second, int)):
13             raise ValueError("Оба значения должны быть целыми числами.")
14
15         if firts >= second:
16             raise ValueError(
17                 "Значение first должно быть меньше значения second."
18             )
19
20         self.first = firts
21         self.second = second
22
23     # Ввод данных с клавиатуры
24     def read(self) -> bool:
25         try:
26             self.first = int(input("Введите значение для first: "))
27             self.second = int(input("Введите значение для second: "))
28             return True
29         except ValueError:
30             return False
31
32     def __str__(self) -> str:
33         return f"({self.first}, {self.second})"
34
35     def __add__(self, other) -> Pair:
36         return Pair(self.first + other.first, self.second + other.second)
37
38     def __sub__(self, other) -> Pair:
39         return Pair(self.first - other.first, self.second - other.second)
40
41     def __mul__(self, other) -> Pair:
42         return Pair(self.first * other.first, self.second * other.second)
43
44     def __div__(self, other) -> Pair:
45         return Pair(self.first / other.first, self.second / other.second)
46
47     def __eq__(self, other) -> bool:
48         return self.first == other.first and self.second == other.second
49
50     def __lt__(self, other) -> bool:
51         return self.first < other.first and self.second < other.second
52
53     def __gt__(self, other) -> bool:
54         return self.first > other.first and self.second > other.second
55
56     def __le__(self, other) -> bool:
57         return self.first <= other.first and self.second <= other.second
58
59     def __ge__(self, other) -> bool:
60         return self.first >= other.first and self.second >= other.second
61
62     def __ne__(self, other) -> bool:
63         return self.first != other.first or self.second != other.second
64
65     def __hash__(self) -> int:
66         return hash((self.first, self.second))
67
68     def __getitem__(self, index) -> int:
69         if index == 0:
70             return self.first
71         elif index == 1:
72             return self.second
73         else:
74             raise IndexError("Индекс выходит за пределы списка")
75
76     def __setitem__(self, index, value) -> None:
77         if index == 0:
78             self.first = value
79         elif index == 1:
80             self.second = value
81         else:
82             raise IndexError("Индекс выходит за пределы списка")
83
84     def __delitem__(self, index) -> None:
85         if index == 0:
86             self.first = 0
87         elif index == 1:
88             self.second = 0
89         else:
90             raise IndexError("Индекс выходит за пределы списка")
91
92     def __len__(self) -> int:
93         return 2
94
95     def __iter__(self) -> list:
96         return [self.first, self.second]
97
98     def __contains__(self, item) -> bool:
99         return item in [self.first, self.second]
100
101     def __repr__(self) -> str:
102         return f"Pair({self.first}, {self.second})"
103
104     def __copy__(self) -> Pair:
105         return Pair(self.first, self.second)
106
107     def __deepcopy__(self, memo) -> Pair:
108         return Pair(self.first, self.second)
109
110     def __getattr__(self, name) -> None:
111         if name == "first":
112             return self.first
113         elif name == "second":
114             return self.second
115         else:
116             raise AttributeError(f"Атрибут {name} не найден")
117
118     def __setattr__(self, name, value) -> None:
119         if name == "first":
120             self.first = value
121         elif name == "second":
122             self.second = value
123         else:
124             raise AttributeError(f"Атрибут {name} не найден")
125
126     def __delattr__(self, name) -> None:
127         if name == "first":
128             self.first = 0
129         elif name == "second":
130             self.second = 0
131         else:
132             raise AttributeError(f"Атрибут {name} не найден")
133
134     def __call__(self) -> Pair:
135         return Pair(self.first, self.second)
136
137     def __getitem__(self, index) -> int:
138         if index == 0:
139             return self.first
140         elif index == 1:
141             return self.second
142         else:
143             raise IndexError("Индекс выходит за пределы списка")
144
145     def __setitem__(self, index, value) -> None:
146         if index == 0:
147             self.first = value
148         elif index == 1:
149             self.second = value
150         else:
151             raise IndexError("Индекс выходит за пределы списка")
152
153     def __delitem__(self, index) -> None:
154         if index == 0:
155             self.first = 0
156         elif index == 1:
157             self.second = 0
158         else:
159             raise IndexError("Индекс выходит за пределы списка")
160
161     def __len__(self) -> int:
162         return 2
163
164     def __iter__(self) -> list:
165         return [self.first, self.second]
166
167     def __contains__(self, item) -> bool:
168         return item in [self.first, self.second]
169
170     def __repr__(self) -> str:
171         return f"Pair({self.first}, {self.second})"
172
173     def __copy__(self) -> Pair:
174         return Pair(self.first, self.second)
175
176     def __deepcopy__(self, memo) -> Pair:
177         return Pair(self.first, self.second)
178
179     def __getattr__(self, name) -> None:
180         if name == "first":
181             return self.first
182         elif name == "second":
183             return self.second
184         else:
185             raise AttributeError(f"Атрибут {name} не найден")
186
187     def __setattr__(self, name, value) -> None:
188         if name == "first":
189             self.first = value
190         elif name == "second":
191             self.second = value
192         else:
193             raise AttributeError(f"Атрибут {name} не найден")
194
195     def __delattr__(self, name) -> None:
196         if name == "first":
197             self.first = 0
198         elif name == "second":
199             self.second = 0
200         else:
201             raise AttributeError(f"Атрибут {name} не найден")
202
203     def __call__(self) -> Pair:
204         return Pair(self.first, self.second)
205
206     def __getitem__(self, index) -> int:
207         if index == 0:
208             return self.first
209         elif index == 1:
210             return self.second
211         else:
212             raise IndexError("Индекс выходит за пределы списка")
213
214     def __setitem__(self, index, value) -> None:
215         if index == 0:
216             self.first = value
217         elif index == 1:
218             self.second = value
219         else:
220             raise IndexError("Индекс выходит за пределы списка")
221
222     def __delitem__(self, index) -> None:
223         if index == 0:
224             self.first = 0
225         elif index == 1:
226             self.second = 0
227         else:
228             raise IndexError("Индекс выходит за пределы списка")
229
230     def __len__(self) -> int:
231         return 2
232
233     def __iter__(self) -> list:
234         return [self.first, self.second]
235
236     def __contains__(self, item) -> bool:
237         return item in [self.first, self.second]
238
239     def __repr__(self) -> str:
240         return f"Pair({self.first}, {self.second})"
241
242     def __copy__(self) -> Pair:
243         return Pair(self.first, self.second)
244
245     def __deepcopy__(self, memo) -> Pair:
246         return Pair(self.first, self.second)
247
248     def __getattr__(self, name) -> None:
249         if name == "first":
250             return self.first
251         elif name == "second":
252             return self.second
253         else:
254             raise AttributeError(f"Атрибут {name} не найден")
255
256     def __setattr__(self, name, value) -> None:
257         if name == "first":
258             self.first = value
259         elif name == "second":
260             self.second = value
261         else:
262             raise AttributeError(f"Атрибут {name} не найден")
263
264     def __delattr__(self, name) -> None:
265         if name == "first":
266             self.first = 0
267         elif name == "second":
268             self.second = 0
269         else:
270             raise AttributeError(f"Атрибут {name} не найден")
271
272     def __call__(self) -> Pair:
273         return Pair(self.first, self.second)
274
275     def __getitem__(self, index) -> int:
276         if index == 0:
277             return self.first
278         elif index == 1:
279             return self.second
280         else:
281             raise IndexError("Индекс выходит за пределы списка")
282
283     def __setitem__(self, index, value) -> None:
284         if index == 0:
285             self.first = value
286         elif index == 1:
287             self.second = value
288         else:
289             raise IndexError("Индекс выходит за пределы списка")
290
291     def __delitem__(self, index) -> None:
292         if index == 0:
293             self.first = 0
294         elif index == 1:
295             self.second = 0
296         else:
297             raise IndexError("Индекс выходит за пределы списка")
298
299     def __len__(self) -> int:
300         return 2
301
302     def __iter__(self) -> list:
303         return [self.first, self.second]
304
305     def __contains__(self, item) -> bool:
306         return item in [self.first, self.second]
307
308     def __repr__(self) -> str:
309         return f"Pair({self.first}, {self.second})"
310
311     def __copy__(self) -> Pair:
312         return Pair(self.first, self.second)
313
314     def __deepcopy__(self, memo) -> Pair:
315         return Pair(self.first, self.second)
316
317     def __getattr__(self, name) -> None:
318         if name == "first":
319             return self.first
320         elif name == "second":
321             return self.second
322         else:
323             raise AttributeError(f"Атрибут {name} не найден")
324
325     def __setattr__(self, name, value) -> None:
326         if name == "first":
327             self.first = value
328         elif name == "second":
329             self.second = value
330         else:
331             raise AttributeError(f"Атрибут {name} не найден")
332
333     def __delattr__(self, name) -> None:
334         if name == "first":
335             self.first = 0
336         elif name == "second":
337             self.second = 0
338         else:
339             raise AttributeError(f"Атрибут {name} не найден")
340
341     def __call__(self) -> Pair:
342         return Pair(self.first, self.second)
343
344     def __getitem__(self, index) -> int:
345         if index == 0:
346             return self.first
347         elif index == 1:
348             return self.second
349         else:
350             raise IndexError("Индекс выходит за пределы списка")
351
352     def __setitem__(self, index, value) -> None:
353         if index == 0:
354             self.first = value
355         elif index == 1:
356             self.second = value
357         else:
358             raise IndexError("Индекс выходит за пределы списка")
359
360     def __delitem__(self, index) -> None:
361         if index == 0:
362             self.first = 0
363         elif index == 1:
364             self.second = 0
365         else:
366             raise IndexError("Индекс выходит за пределы списка")
367
368     def __len__(self) -> int:
369         return 2
370
371     def __iter__(self) -> list:
372         return [self.first, self.second]
373
374     def __contains__(self, item) -> bool:
375         return item in [self.first, self.second]
376
377     def __repr__(self) -> str:
378         return f"Pair({self.first}, {self.second})"
379
380     def __copy__(self) -> Pair:
381         return Pair(self.first, self.second)
382
383     def __deepcopy__(self, memo) -> Pair:
384         return Pair(self.first, self.second)
385
386     def __getattr__(self, name) -> None:
387         if name == "first":
388             return self.first
389         elif name == "second":
390             return self.second
391         else:
392             raise AttributeError(f"Атрибут {name} не найден")
393
394     def __setattr__(self, name, value) -> None:
395         if name == "first":
396             self.first = value
397         elif name == "second":
398             self.second = value
399         else:
400             raise AttributeError(f"Атрибут {name} не найден")
401
402     def __delattr__(self, name) -> None:
403         if name == "first":
404             self.first = 0
405         elif name == "second":
406             self.second = 0
407         else:
408             raise AttributeError(f"Атрибут {name} не найден")
409
410     def __call__(self) -> Pair:
411         return Pair(self.first, self.second)
412
413     def __getitem__(self, index) -> int:
414         if index == 0:
415             return self.first
416         elif index == 1:
417             return self.second
418         else:
419             raise IndexError("Индекс выходит за пределы списка")
420
421     def __setitem__(self, index, value) -> None:
422         if index == 0:
423             self.first = value
424         elif index == 1:
425             self.second = value
426         else:
427             raise IndexError("Индекс выходит за пределы списка")
428
429     def __delitem__(self, index) -> None:
430         if index == 0:
431             self.first = 0
432         elif index == 1:
433             self.second = 0
434         else:
435             raise IndexError("Индекс выходит за пределы списка")
436
437     def __len__(self) -> int:
438         return 2
439
440     def __iter__(self) -> list:
441         return [self.first, self.second]
442
443     def __contains__(self, item) -> bool:
444         return item in [self.first, self.second]
445
446     def __repr__(self) -> str:
447         return f"Pair({self.first}, {self.second})"
448
449     def __copy__(self) -> Pair:
450         return Pair(self.first, self.second)
451
452     def __deepcopy__(self, memo) -> Pair:
453         return Pair(self.first, self.second)
454
455     def __getattr__(self, name) -> None:
456         if name == "first":
457             return self.first
458         elif name == "second":
459             return self.second
460         else:
461             raise AttributeError(f"Атрибут {name} не найден")
462
463     def __setattr__(self, name, value) -> None:
464         if name == "first":
465             self.first = value
466         elif name == "second":
467             self.second = value
468         else:
469             raise AttributeError(f"Атрибут {name} не найден")
470
471     def __delattr__(self, name) -> None:
472         if name == "first":
473             self.first = 0
474         elif name == "second":
475             self.second = 0
476         else:
477             raise AttributeError(f"Атрибут {name} не найден")
478
479     def __call__(self) -> Pair:
480         return Pair(self.first, self.second)
481
482     def __getitem__(self, index) -> int:
483         if index == 0:
484             return self.first
485         elif index == 1:
486             return self.second
487         else:
488             raise IndexError("Индекс выходит за пределы списка")
489
490     def __setitem__(self, index, value) -> None:
491         if index == 0:
492             self.first = value
493         elif index == 1:
494             self.second = value
495         else:
496             raise IndexError("Индекс выходит за пределы списка")
497
498     def __delitem__(self, index) -> None:
499         if index == 0:
500             self.first = 0
501         elif index == 1:
502             self.second = 0
503         else:
504             raise IndexError("Индекс выходит за пределы списка")
505
506     def __len__(self) -> int:
507         return 2
508
509     def __iter__(self) -> list:
510         return [self.first, self.second]
511
512     def __contains__(self, item) -> bool:
513         return item in [self.first, self.second]
514
515     def __repr__(self) -> str:
516         return f"Pair({self.first}, {self.second})"
517
518     def __copy__(self) -> Pair:
519         return Pair(self.first, self.second)
520
521     def __deepcopy__(self, memo) -> Pair:
522         return Pair(self.first, self.second)
523
524     def __getattr__(self, name) -> None:
525         if name == "first":
526             return self.first
527         elif name == "second":
528             return self.second
529         else:
530             raise AttributeError(f"Атрибут {name} не найден")
531
532     def __setattr__(self, name, value) -> None:
533         if name == "first":
534             self.first = value
535         elif name == "second":
536             self.second = value
537         else:
538             raise AttributeError(f"Атрибут {name} не найден")
539
540     def __delattr__(self, name) -> None:
541         if name == "first":
542             self.first = 0
543         elif name == "second":
544             self.second = 0
545         else:
546             raise AttributeError(f"Атрибут {name} не найден")
547
548     def __call__(self) -> Pair:
549         return Pair(self.first, self.second)
550
551     def __getitem__(self, index) -> int:
552         if index == 0:
553             return self.first
554         elif index == 1:
555             return self.second
556         else:
557             raise IndexError("Индекс выходит за пределы списка")
558
559     def __setitem__(self, index, value) -> None:
560         if index == 0:
561             self.first = value
562         elif index == 1:
563             self.second = value
564         else:
565             raise IndexError("Индекс выходит за пределы списка")
566
567     def __delitem__(self, index) -> None:
568         if index == 0:
569             self.first = 0
570         elif index == 1:
571             self.second = 0
572         else:
573             raise IndexError("Индекс выходит за пределы списка")
574
575     def __len__(self) -> int:
576         return 2
577
578     def __iter__(self) -> list:
579         return [self.first, self.second]
580
581     def __contains__(self, item) -> bool:
582         return item in [self.first, self.second]
583
584     def __repr__(self) -> str:
585         return f"Pair({self.first}, {self.second})"
586
587     def __copy__(self) -> Pair:
588         return Pair(self.first, self.second)
589
590     def __deepcopy__(self, memo) -> Pair:
591         return Pair(self.first, self.second)
592
593     def __getattr__(self, name) -> None:
594         if name == "first":
595             return self.first
596         elif name == "second":
597             return self.second
598         else:
599             raise AttributeError(f"Атрибут {name} не найден")
600
601     def __setattr__(self, name, value) -> None:
602         if name == "first":
603             self.first = value
604         elif name == "second":
605             self.second = value
606         else:
607             raise AttributeError(f"Атрибут {name} не найден")
608
609     def __delattr__(self, name) -> None:
610         if name == "first":
611             self.first = 0
612         elif name == "second":
613             self.second = 0
614         else:
615             raise AttributeError(f"Атрибут {name} не найден")
616
617     def __call__(self) -> Pair:
618         return Pair(self.first, self.second)
619
620     def __getitem__(self, index) -> int:
621         if index == 0:
622             return self.first
623         elif index == 1:
624             return self.second
625         else:
626             raise IndexError("Индекс выходит за пределы списка")
627
628     def __setitem__(self, index, value) -> None:
629         if index == 0:
630             self.first = value
631         elif index == 1:
632             self.second = value
633         else:
634             raise IndexError("Индекс выходит за пределы списка")
635
636     def __delitem__(self, index) -> None:
637         if index == 0:
638             self.first = 0
639         elif index == 1:
640             self.second = 0
641         else:
642             raise IndexError("Индекс выходит за пределы списка")
643
644     def __len__(self) -> int:
645         return 2
646
647     def __iter__(self) -> list:
648         return [self.first, self.second]
649
650     def __contains__(self, item) -> bool:
651         return item in [self.first, self.second]
652
653     def __repr__(self) -> str:
654         return f"Pair({self.first}, {self.second})"
655
656     def __copy__(self) -> Pair:
657         return Pair(self.first, self.second)
658
659     def __deepcopy__(self, memo) -> Pair:
660         return Pair(self.first, self.second)
661
662     def __getattr__(self, name) -> None:
663         if name == "first":
664             return self.first
665         elif name == "second":
666             return self.second
667         else:
668             raise AttributeError(f"Атрибут {name} не найден")
669
670     def __setattr__(self, name, value) -> None:
671         if name == "first":
672             self.first = value
673         elif name == "second":
674             self.second = value
675         else:
676             raise AttributeError(f"Атрибут {name} не найден")
677
678     def __delattr__(self, name) -> None:
679         if name == "first":
680             self.first = 0
681         elif name == "second":
682             self.second = 0
683         else:
684             raise AttributeError(f"Атрибут {name} не найден")
685
686     def __call__(self) -> Pair:
687         return Pair(self.first, self.second)
688
689     def __getitem__(self, index) -> int:
690         if index == 0:
691             return self.first
692         elif index == 1:
693             return self.second
694         else:
695             raise IndexError("Индекс выходит за пределы списка")
696
697     def __setitem__(self, index, value) -> None:
698         if index == 0:
699             self.first = value
700         elif index == 1:
701             self.second = value
702         else:
703             raise IndexError("Индекс выходит за пределы списка")
704
705     def __delitem__(self, index) -> None:
706         if index == 0:
707             self.first = 0
708         elif index == 1:
709             self.second = 0
710         else:
711             raise IndexError("Индекс выходит за пределы списка")
712
713     def __len__(self) -> int:
714         return 2
715
716     def __iter__(self) -> list:
717         return [self.first, self.second]
718
719     def __contains__(self, item) -> bool:
720         return item in [self.first, self.second]
721
722     def __repr__(self) -> str:
723         return f"Pair({self.first}, {self.second})"
724
725     def __copy__(self) -> Pair:
726         return Pair(self.first, self.second)
727
728     def __deepcopy__(self, memo) -> Pair:
729         return Pair(self.first, self.second)
730
731     def __getattr__(self, name) -> None:
732         if name == "first":
733             return self.first
734         elif name == "second":
735             return self.second
736         else:
737             raise AttributeError(f"Атрибут {name} не найден")
738
739     def __setattr__(self, name, value) -> None:
740         if name == "first":
741             self.first = value
742         elif name == "second":
743             self.second = value
744         else:
745             raise AttributeError(f"Атрибут {name} не найден")
746
747     def __delattr__(self, name) -> None:
748         if name == "first":
749             self.first = 0
750         elif name == "second":
751             self.second = 0
752         else:
753             raise AttributeError(f"Атрибут {name} не найден")
754
755     def __call__(self) -> Pair:
756         return Pair(self.first, self.second)
757
758     def __getitem__(self, index) -> int:
759         if index == 0:
760             return self.first
761         elif index == 1:
762             return self.second
763         else:
764             raise IndexError("Индекс выходит за пределы списка")
765
766     def __setitem__(self, index, value) -> None:
767         if index == 0:
768             self.first = value
769         elif index == 1:
770             self.second = value
771         else:
772             raise IndexError("Индекс выходит за пределы списка")
773
774     def __delitem__(self, index) -> None:
775         if index == 0:
776             self.first = 0
777         elif index == 1:
778             self.second = 0
779         else:
780             raise IndexError("Индекс выходит за пределы списка")
781
782     def __len__(self) -> int:
783         return 2
784
785     def __iter__(self) -> list:
786         return [self.first, self.second]
787
788     def __contains__(self, item) -> bool:
789         return item in [self.first, self.second]
790
791     def __repr__(self) -> str:
792         return f"Pair({self.first}, {self.second})"
793
794     def __copy__(self) -> Pair:
795         return Pair(self.first, self.second)
796
797     def __deepcopy__(self, memo) -> Pair:
798         return Pair(self.first, self.second)
799
800     def __getattr__(self, name) -> None:
801         if name == "first":
802             return self.first
803         elif name == "second":
804             return self.second
805         else:
806             raise AttributeError(f"Атрибут {name} не найден")
807
808     def __setattr__(self, name, value) -> None:
809         if name == "first":
810             self.first = value
811         elif name == "second":
812             self.second = value
813         else:
814             raise AttributeError(f"Атрибут {name} не найден")
815
816     def __delattr__(self, name) -> None:
817         if name == "first":
818             self.first = 0
819         elif name == "second":
820             self.second = 0
821         else:
822             raise AttributeError(f"Атрибут {name} не найден")
823
824     def __call__(self) -> Pair:
825         return Pair(self.first, self.second)
826
827     def __getitem__(self, index) -> int:
828         if index == 0:
829             return self.first
830         elif index == 1:
831             return self.second
832         else:
833             raise IndexError("Индекс выходит за пределы списка")
834
835     def __setitem__(self, index, value) -> None:
836         if index == 0:
837             self.first = value
838         elif index == 1:
839             self.second = value
840         else:
841             raise IndexError("Индекс выходит за пределы списка")
842
843     def __delitem__(self, index) -> None:
844         if index == 0:
845             self.first = 0
846         elif index == 1:
847             self.second = 0
848         else:
849             raise IndexError("Индекс выходит за пределы списка")
850
851     def __len__(self) -> int:
852         return 2
853
854     def __iter__(self) -> list:
855         return [self.first, self.second]
856
857     def __contains__(self, item) -> bool:
858         return item in [self.first, self.second]
859
860     def __repr__(self) -> str:
861         return f"Pair({self.first}, {self.second})"
862
863     def __copy__(self) -> Pair:
864         return Pair(self.first, self.second)
865
866     def __deepcopy__(self, memo) -> Pair:
867         return Pair(self.first, self.second)
868
869     def __getattr__(self, name) -> None:
870         if name == "first":
871             return self.first
872         elif name == "second":
873             return self.second
874         else:
875             raise AttributeError(f"Атрибут {name} не найден")
876
877     def __setattr__(self, name, value) -> None:
878         if name == "first":
879             self.first = value
880         elif name == "second":
881             self.second = value
882         else:
883             raise AttributeError(f"Атрибут {name} не найден")
884
885     def __delattr__(self, name) -> None:
886         if name == "first":
887             self.first = 0
888         elif name == "second":
889             self.second = 0
890         else:
891             raise AttributeError(f"Атрибут {name} не найден")
892
893     def __call__(self) -> Pair:
894         return Pair(self.first, self.second)
895
896     def __getitem__(self, index) -> int:
897         if index == 0:
898             return self.first
899         elif index == 1:
900             return self.second
901         else:
902             raise IndexError("Индекс выходит за пределы списка")
903
904     def __setitem__(self, index, value) -> None:
905         if index == 0:
906             self.first = value
907         elif index == 1:
908             self.second = value
909         else:
910             raise IndexError("Индекс выходит за пределы списка")
911
912     def __delitem__(self, index) -> None:
913         if index == 0:
914             self.first = 0
915         elif index == 1:
916             self.second = 0
917         else:
918             raise IndexError("Индекс выходит за пределы списка")
919
920     def __len__(self) -> int:
921         return 2
922
923     def __iter__(self) -> list:
924         return [self.first, self.second]
925
926     def __contains__(self, item) -> bool:
927         return item in [self.first, self.second]
928
929     def __repr__(self) -> str:
930         return f"Pair({self.first}, {self.second})"
931
932     def __copy__(self) -> Pair:
933         return Pair(self.first, self.second)
934
935     def __deepcopy__(self, memo) -> Pair:
936         return Pair(self.first, self.second)
937
938     def __getattr__(self, name) -> None:
939         if name == "first":
940             return self.first
941         elif name == "second":
942             return self.second
943         else:
944             raise AttributeError(f"Атрибут {name} не найден")
945
946     def __setattr__(self, name, value) -> None:
947         if name == "first":
948             self.first = value
949         elif name == "second":
950             self.second = value
951         else:
952             raise AttributeError(f"Атрибут {name} не найден")
953
954     def __delattr__(self, name) -> None:
955         if name == "first":
956             self.first = 0
957         elif name == "second":
958             self.second = 0
959         else:
960             raise AttributeError(f"Атрибут {name} не найден")
961
962     def __call__(self) -> Pair:
963         return Pair(self.first, self.second)
964
965     def __getitem__(self, index) -> int:
966         if index == 0:
967             return self.first
968         elif index == 1:
969             return self.second
970         else:
971             raise IndexError("Индекс выходит за пределы списка")
972
973     def __setitem__(self, index, value) -> None:
974         if index == 0:
975             self.first = value
976         elif index == 1:
977             self.second = value
978         else:
979             raise IndexError("Индекс выходит за пределы списка")
980
981     def __delitem__(self, index) -> None:
982         if index == 0:
983             self.first = 0
984         elif index == 1:
985             self.second = 0
986         else:
987             raise IndexError("Индекс выходит за пределы списка")
988
989     def __len__(self) -> int:
990         return 2
991
992     def __iter__(self) -> list:
993         return [self.first, self.second]
994
995     def __contains__(self, item) -> bool:
996         return item in [self.first, self.second]
997
998     def __repr__(self) -> str:
999         return f"Pair({self.first}, {self.second})"
1000
1001     def __copy__(self) -> Pair:
1002         return Pair(self.first, self.second)
1003
1004     def __deepcopy__(self, memo) -> Pair:
1005         return Pair(self.first, self.second)
1006
1007     def __getattr__(self, name) -> None:
1008         if name == "first":
1009             return self.first
1010         elif name == "second":
1011             return self.second
1012         else:
1013             raise AttributeError(f"Атрибут {name} не найден")
1014
1015     def __setattr__(self, name, value) -> None:
1016         if name == "first":
1017             self.first = value
1018         elif name == "second":
1019             self.second = value
1020         else:
1021             raise AttributeError(f"Атрибут {name} не найден")
1022
1023     def __delattr__(self, name) -> None:
1024         if name == "first":
1025             self.first = 0
1026         elif name == "second":
1027             self.second = 0
1028         else:
1029             raise AttributeError(f"Атрибут {name} не найден")
1030
1031     def __call__(self) -> Pair:
1032         return Pair(self.first, self.second)
1033
1034     def __getitem__(self, index) -> int:
1035         if index == 0:
1036             return self.first
1037         elif index == 1:
1038             return self.second
1039         else:
1040             raise IndexError("Индекс выходит за пределы списка")
1041
1042     def __setitem__(self, index, value) -> None:
1043         if index == 0:
1044             self.first = value
1045         elif index == 1:
1046             self.second = value
1047         else:
1048             raise IndexError("Индекс выходит за пределы списка")
1049
1050     def __delitem__(self, index) -> None:
1051         if index == 0:
1052             self.first = 0
1053         elif index == 1:
1054             self.second = 0
1055         else:
1056             raise IndexError("Индекс выходит за пределы списка")
1057
1058     def __len__(self) -> int:
1059         return 2
1060
1061     def __iter__(self) -> list:
1062         return [self.first, self.second]
1063
1064     def __contains__(self, item) -> bool:
1065         return item in [self.first, self.second]
1066
1067     def __repr__(self) -> str:
1068         return f"Pair({self.first}, {self.second})"
1069
1070     def __copy__(self) -> Pair:
1071         return Pair(self.first, self.second)
1072
1073     def __deepcopy__(self, memo) -> Pair:
1074         return Pair(self.first, self.second)
1075
1076     def __getattr__(self, name) -> None:
1077         if name == "first":
1078             return self.first
1079         elif name == "second":
1080             return self.second
1081         else:
1082             raise AttributeError(f"Атрибут {name} не найден")
1083
1084     def __setattr__(self, name, value) -> None:
1085         if name == "first":
1086             self.first = value
1087         elif name == "second":
108
```

```

> .venv
  examples
    example_1.py U
    example_2.py U
    example_3.py U
  src
    individual_task_1.... U
    individual_task_2.... U
  .gitignore
  .pre-commit-config... U
  LICENSE
  README.md
  requirements.txt U

class Money:
    def __init__(self, initial_data=None, max_size=10):
        self.size = max_size # Максимальное количество элементов
        self.data = [] # Список словарей для представления купюр
        self.count = 0 # Текущее количество элементов

        # Инициализация через строку или список словарей
        if isinstance(initial_data, str):
            self._initialize_from_string(initial_data)
        elif isinstance(initial_data, list):
            self._initialize_from_list(initial_data)

    def _initialize_from_string(self, data_string):
        pairs = data_string.split(", ")
        for pair in pairs:
            denom, count = pair.split(":")
            self.add(denom.strip(), int(count.strip()))
        self.count = len(self.data)

    def _initialize_from_list(self, data_list):
        for item in data_list:
            if "denomination" in item and "count" in item:
                self.add(item["denomination"], item["count"])
        self.count = len(self.data)

```

```

~/work/OOP_LR_3 git develop ?4 python src/individual_task_2.py
10: 5, 50: 2, 100: 1
10: 5, 20: 3, 50: 2, 100: 1
5
4
{'denomination': '100', 'count': 1}

```

7. Зафиксируйте сделанные изменения в репозитории.
8. Выполните слияние ветки для разработки с веткой main / master.

```

~/work/OOP_LR_3 git develop git switch main
Переключились на ветку «main»
Эта ветка соответствует «origin/main».
~/work/OOP_LR_3 git merge develop
Обновление 8beb98b..9fc7c41
Fast-forward
 .pre-commit-config.yaml | 20 +++++
 examples/example_1.py   | 8 +
 examples/example_2.py   | 54 +++++
 examples/example_3.py   | 184 +++++
 requirements.txt         | 1 +
 src/individual_task_1.py | 92 +++++
 src/individual_task_2.py | 88 +++++
 tests/test_money.py     | 79 +++++
 tests/test_pair.py      | 96 +++++
 9 files changed, 622 insertions(+)
 create mode 100644 .pre-commit-config.yaml
 create mode 100644 examples/example_1.py
 create mode 100644 examples/example_2.py
 create mode 100644 examples/example_3.py
 create mode 100644 requirements.txt
 create mode 100644 src/individual_task_1.py
 create mode 100644 src/individual_task_2.py
 create mode 100644 tests/test_money.py
 create mode 100644 tests/test_pair.py

```

9. Отправьте сделанные изменения на сервер GitHub.

```
~/work/OOP_LR_3 main *4 git push
Перечисление объектов: 21, готово.
Подсчет объектов: 100% (21/21), готово.
При сжатии изменений используется до 12 потоков
Сжатие объектов: 100% (19/19), готово.
Запись объектов: 100% (20/20), 6.96 КиБ | 2.32 МиБ/с, готово.
Total 20 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/m4g0med0v/OOP_LR_3.git
8beb98b..9fc7c41 main -> main
```

Контрольные вопросы

1. Какие средства существуют в Python для перегрузки операций?

В Python перегрузка операций осуществляется с помощью специальных методов (магических методов или dunder-методов), которые имеют двойное подчеркивание в начале и конце названия. Эти методы позволяют изменить поведение объектов при использовании стандартных операторов, таких как арифметические операторы, операторы сравнения, индексирование и т. д. Примеры таких методов:

- Арифметические операции: `__add__`, `__sub__`, `__mul__`, `__truediv__`, и другие.
- Операции сравнения: `__eq__`, `__lt__`, `__gt__`, `__le__`, `__ge__`, `__ne__`.
- Индексирование: `__getitem__`, `__setitem__`, `__delitem__`.
- Преобразование типов: `__str__`, `__repr__`, `__int__`, `__float__`.
- Контекстные менеджеры: `__enter__`, `__exit__`.
- Итерации: `__iter__`, `__next__`.

2. Какие существуют методы для перегрузки арифметических операций и операций отношения в языке Python?

Для перегрузки арифметических операций используются следующие методы:

- `__add__(self, other)`: перегрузка оператора +
- `__sub__(self, other)`: перегрузка оператора -
- `__mul__(self, other)`: перегрузка оператора *
- `__truediv__(self, other)`: перегрузка оператора /

- `__floordiv__(self, other)`: перегрузка оператора `//`
- `__mod__(self, other)`: перегрузка оператора `%`
- `__pow__(self, other)`: перегрузка оператора `**`
- `__neg__(self)`: унарный минус (`-self`)
- `__pos__(self)`: унарный плюс (`+self`)
- `__abs__(self)`: абсолютное значение (`abs(self)`)

Для перегрузки операций отношения используются следующие методы:

- `__eq__(self, other)`: перегрузка оператора `==`
- `__ne__(self, other)`: перегрузка оператора `!=`
- `__lt__(self, other)`: перегрузка оператора `<`
- `__le__(self, other)`: перегрузка оператора `<=`
- `__gt__(self, other)`: перегрузка оператора `>`
- `__ge__(self, other)`: перегрузка оператора `>=`

3. В каких случаях будут вызваны следующие методы: `__add__`, `__iadd__` и `__radd__`? Приведите примеры.

`__add__(self, other)`: вызывается при использовании оператора `+` между объектом и другим значением. Например:

```
1 class Number:
2     def __init__(self, value):
3         self.value = value
4
5     def __add__(self, other):
6         return Number(self.value + other)
7
8 num = Number(5)
9 result = num + 10 # вызывает num.__add__(10)
10 print(result.value) # Выведет 15
```

`__iadd__(self, other)`: вызывается при использовании оператора `+=` (операция in-place сложения). Метод `__iadd__` должен изменять объект на месте, если это возможно, и возвращать его. Например:

```

1 class Number:
2     def __init__(self, value):
3         self.value = value
4
5     def __iadd__(self, other):
6         self.value += other
7         return self
8
9 num = Number(5)
10 num += 10 # вызывает num.__iadd__(10)
11 print(num.value) # Выведет 15

```

`__radd__(self, other)`: вызывается, если левый операнд не знает, как выполнить сложение с объектом (например, если у `other` нет метода `__add__`). Например:

```

1 class Number:
2     def __init__(self, value):
3         self.value = value
4
5     def __radd__(self, other):
6         return Number(self.value + other)
7
8 num = Number(5)
9 result = 10 + num # вызывает num.__radd__(10)
10 print(result.value) # Выведет 15

```

4. Для каких целей предназначен метод `__new__` ? Чем он отличается от метода `__init__` ?

`__new__(cls, ...)`: это статический метод, который отвечает за создание нового экземпляра класса. Он вызывается перед `__init__` и непосредственно создаёт объект класса. Метод `__new__` используется, когда нужно контролировать процесс создания экземпляра.

`__init__(self, ...)`: инициализирует объект, который был создан методом `__new__`. Это метод инициализации, в котором можно задавать начальные параметры объекта.

5. Чем отличаются методы `__str__` и `__repr__` ?

`__str__(self)`: возвращает строковое представление объекта, предназначенное для пользователей. Оно должно быть более понятным и удобочитаемым. Этот метод вызывается, например, функцией `print()` или `str()`.

`__repr__(self)`: возвращает строковое представление объекта, которое должно быть максимально точным и однозначным. Оно предназначено для разработчиков и должно, по возможности, содержать такую информацию, чтобы при копировании и выполнении этого представления можно было создать аналогичный объект. Вызывается, например, функцией `repr()` или при выводе объекта в интерактивной консоли.