**Описание полученного задания**

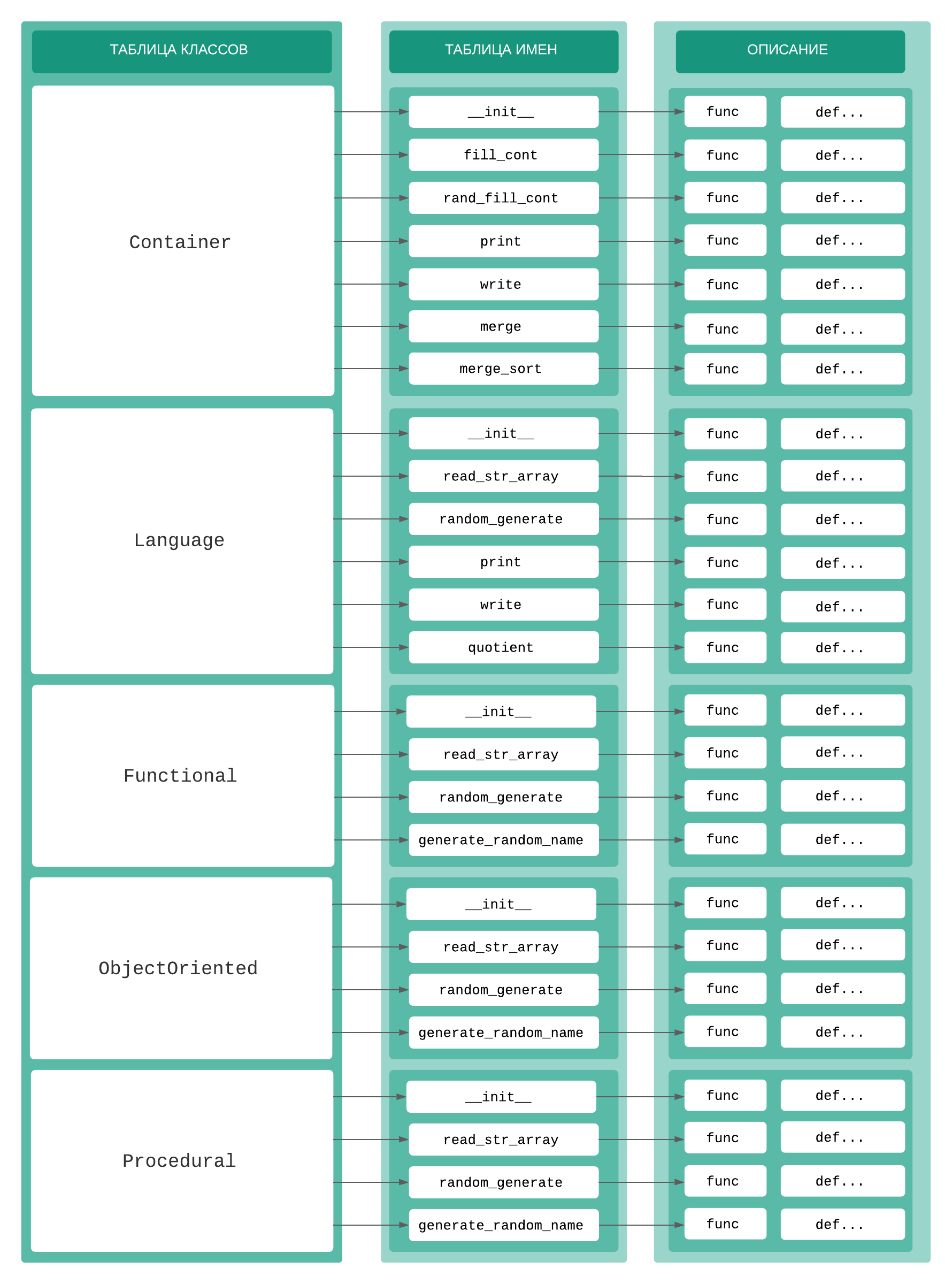
Необходимо написать программу на языке Python, оформленную в виде консольного приложения. Запуск программы осуществляется из командной строки, в которой указываются имя запускаемой программы, имя файла с исходными данными, имя файла с выходными данными.

Для каждого программного объекта, загружаемого в контейнер, исходный файл с тестовым набором должен содержать признак альтернативы, а также список параметров, необходимых альтернативе. В выходной файл необходимо вывести введенные в контейнер данные. Помимо этого, необходимо вывести информацию об общем количестве объектов, содержащихся в контейнере. После этого в тот же файл необходимо вывести новые данные в соответствии с результатами, полученными в ходе работы программы.

Основное задание для составления программы – создание обобщенного артефакта – языки программирования – со следующими базовыми альтернативами, имеющими уникальные параметры: процедурные языки программирования (уникальные параметры: наличие/отсутствие абстрактных типов данных [булевская величина]), объектно-ориентированные языки программирования (уникальные параметры: виды наследования: одинарное, множественное, интерфейс [перечислимый тип]), функциональные языки программирования (уникальные параметры: типизация: строгая, динамическая [перечислимый тип]; поддержка «ленивых» вычислений [булевская величина]). Все базовые альтернативы имеют общие переменные: название [строка символов], популярность в процентах [действительное число], год создания [целое число]. Также все базовые альтернативы имеют общую для всех функцию: частное от деления года создания на количество символов в названии.

После размещения данных в контейнер необходимо упорядочить элементы контейнера по убыванию, используя сортировку с помощью прямого слияния (Straight Merge). В качестве ключей для сортировки и других действий используются результаты функции, общей для всех альтернатив. В программе должны поддерживаться следующие операции: заполнение контейнера данными, поступающими из входного потока (файла); вывод значений всех элементов в выходной поток (файл), выводятся параметры элементов, размещенных в контейнере.

**Отображение содержимого классов**

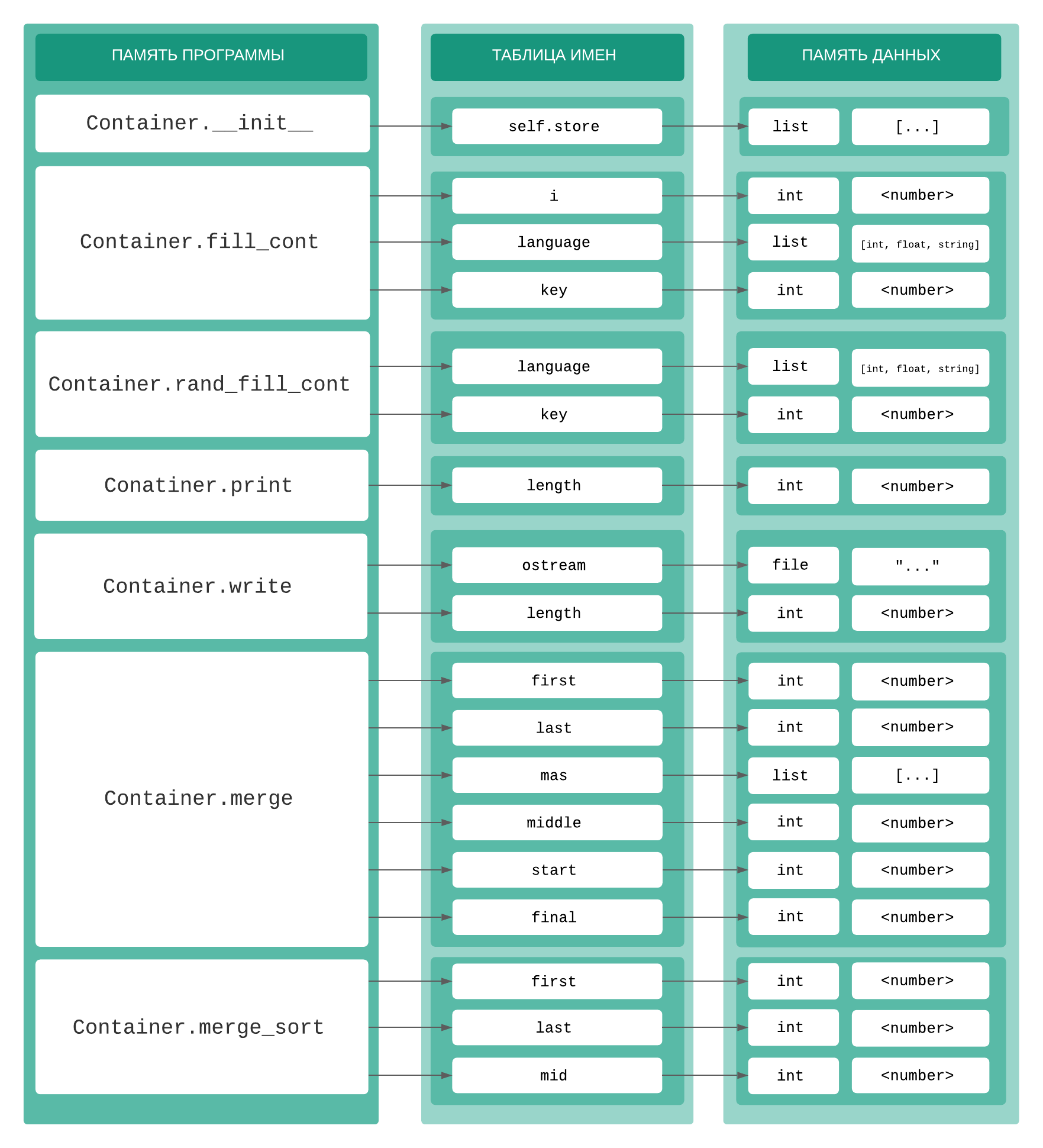


**Отображение на память методов модуля main**

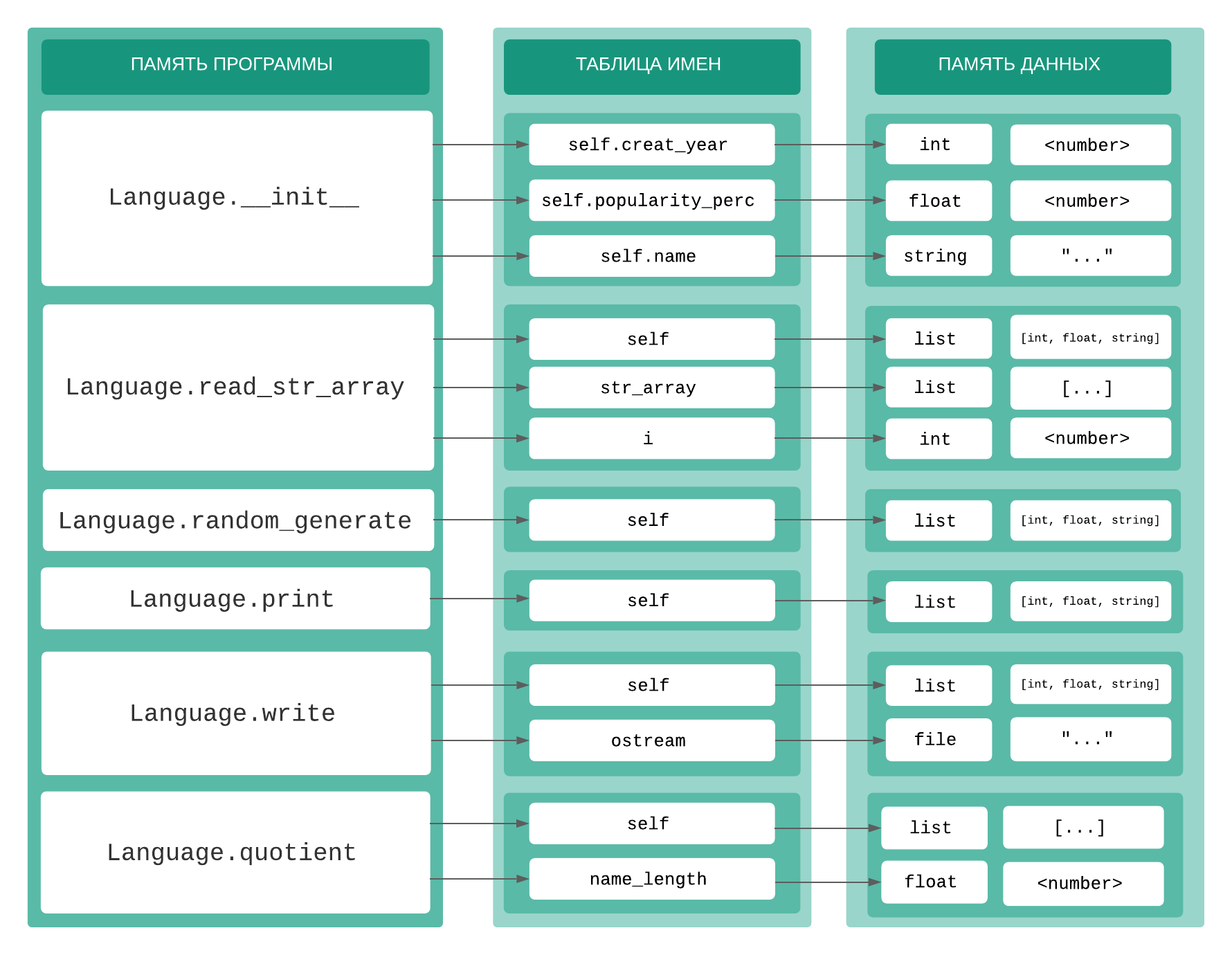
Изображение выглядит как стол

Автоматически созданное описание

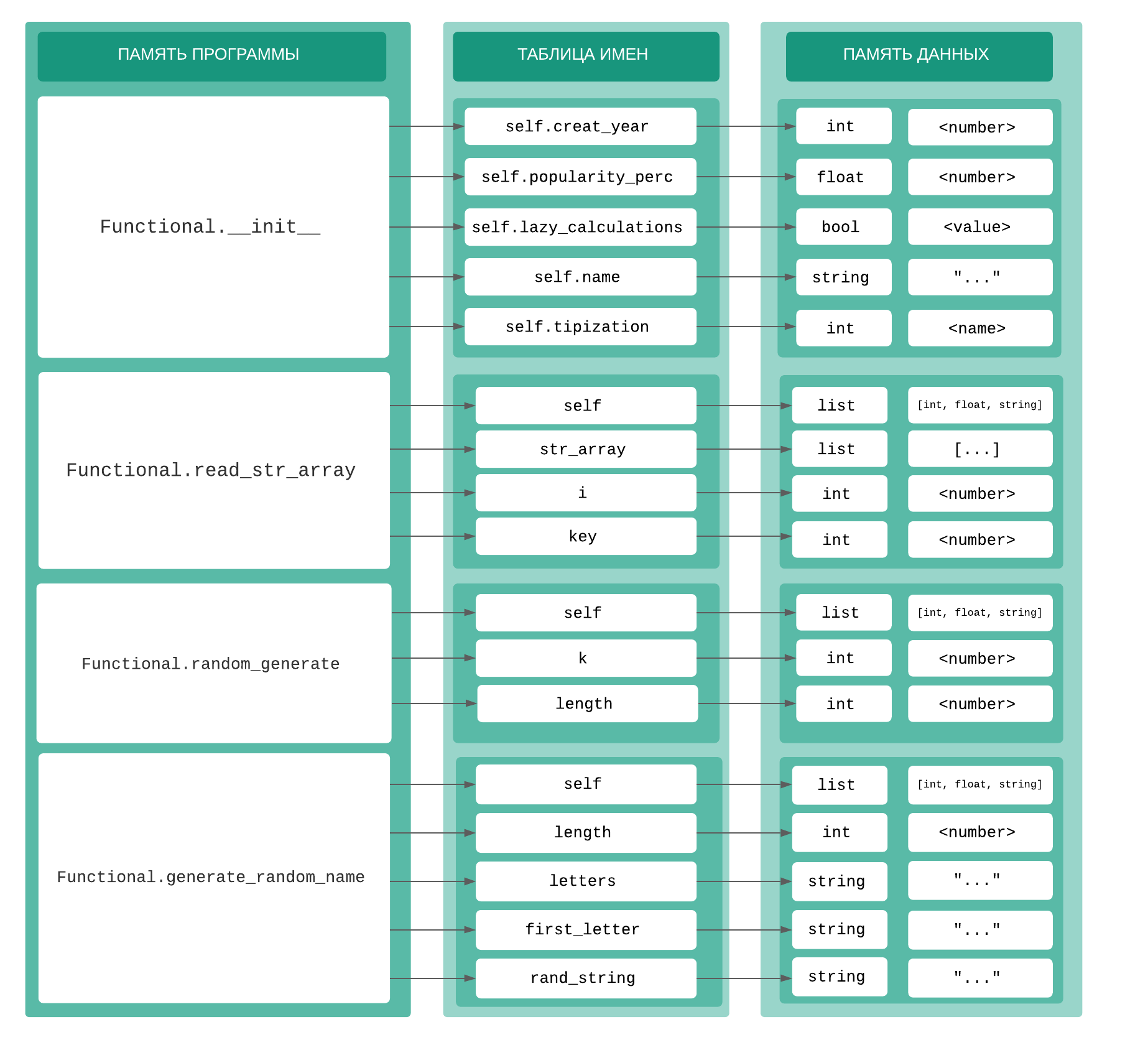
**Отображение на память методов класса Container**



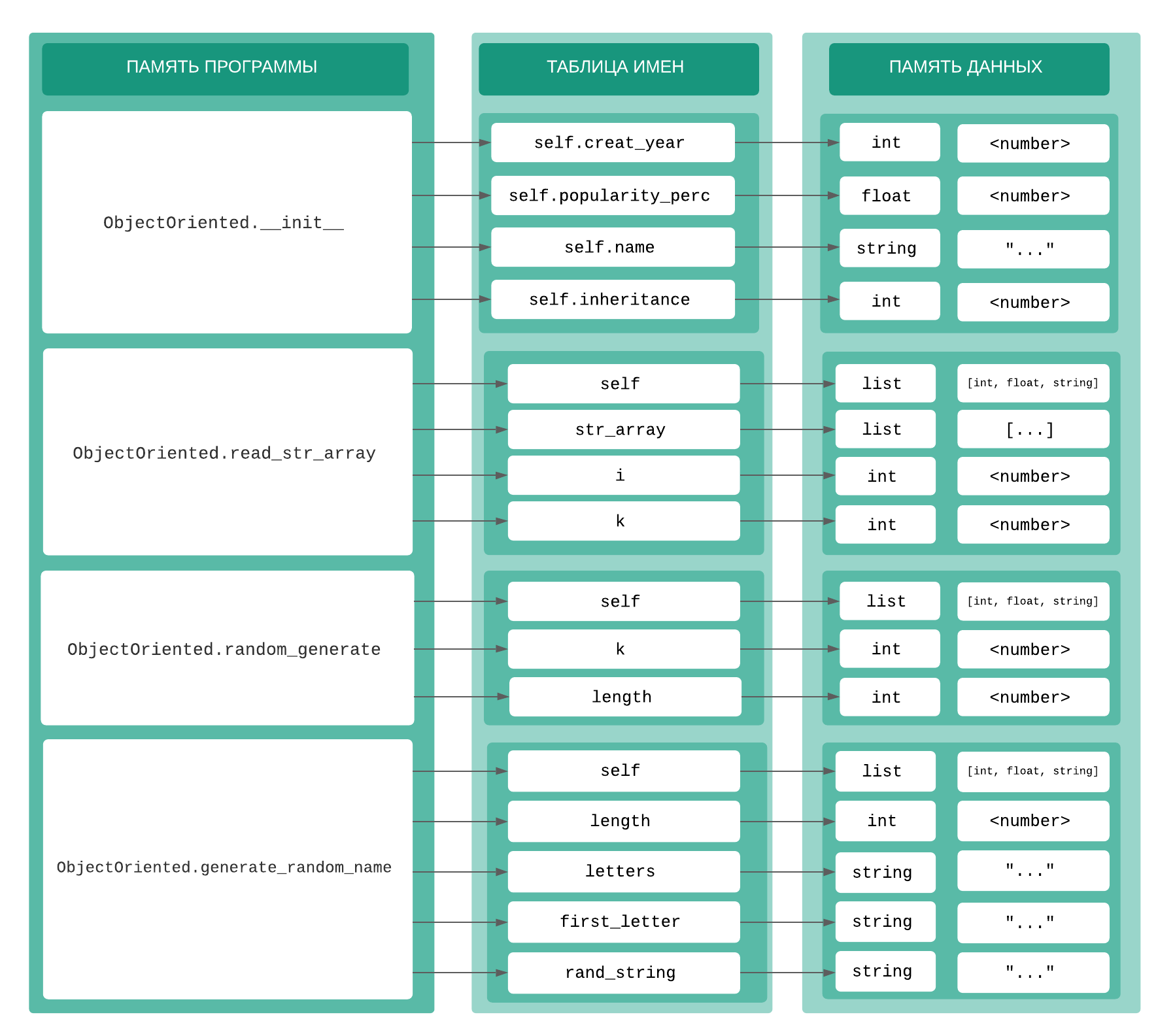
**Отображение на память методов класса Language**



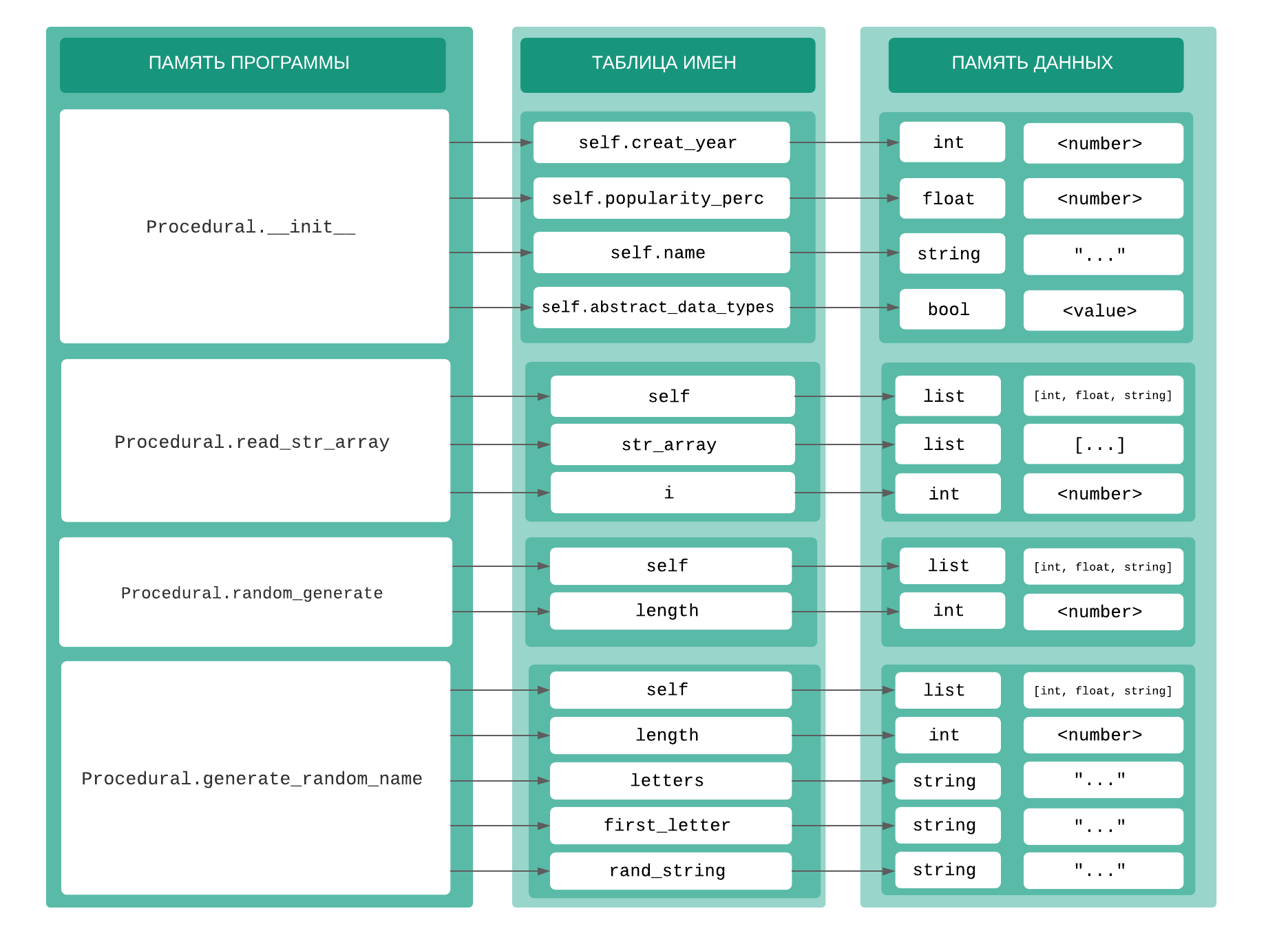
**Отображение на память методов класса Functional**



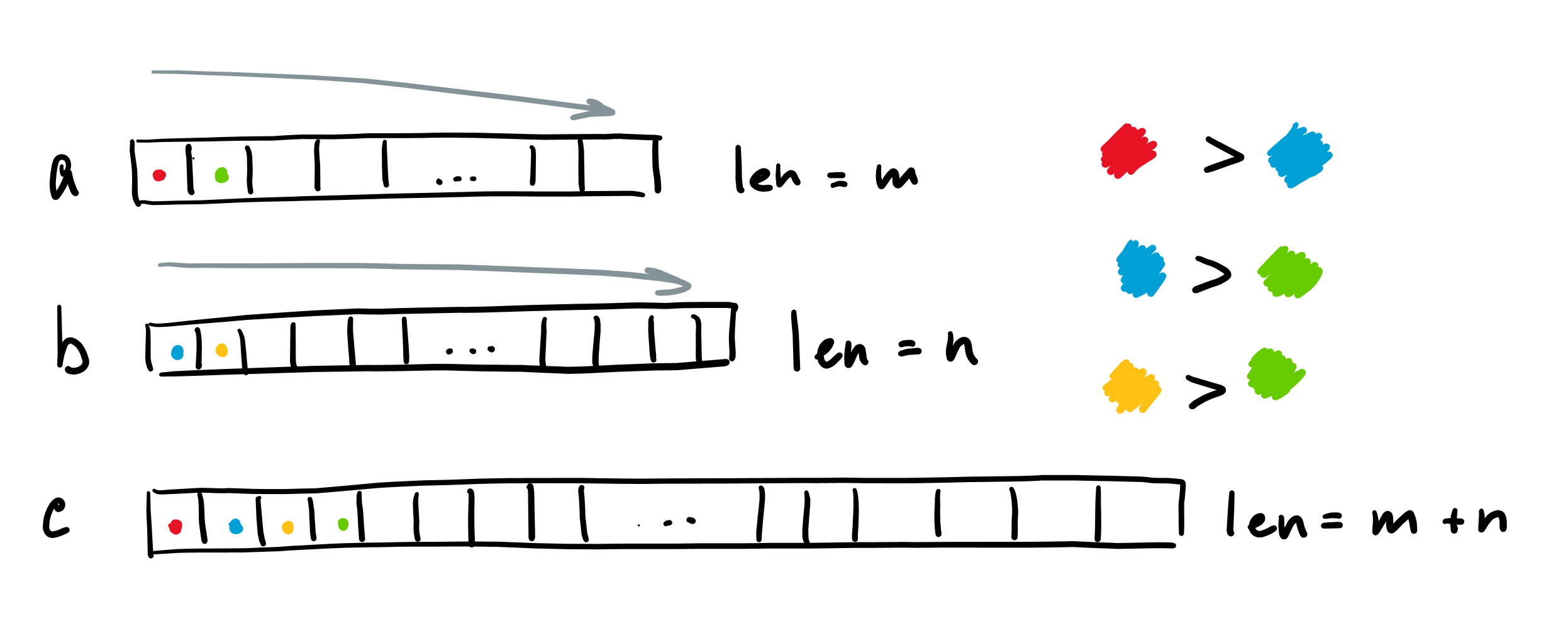
**Отображение на память методов класса ObjectOriented**



**Отображение на память методов класса Procedural**



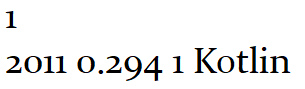
**Принцип алгоритма сортировки контейнера методом слияния:**



**Требуемые метрики, определяющие характеристики программы, для различных тестовых прогонов:**

*Пример* тестовых входных данных:

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**



*Изображение 1*

Изображение выглядит как текст

Автоматически созданное описание

*Изображение 2*

*Замечание*: первым параметром с новой строки (как на изображении 1) или через пробел (как на изображении 2) указывается номер базовой альтернативы в диапазоне от 1 до 3 (1 – процедурный язык программирования; 2 – объектно-ориентированный язык программирования; 3 – функциональный язык программирования). В следующей строке через пробел вводятся параметры альтернативы (уникальные для каждой) в следующем порядке:

1. *процедурный язык программирования*: год создания (целочисленный тип в диапазоне от 1940 до 2020), процент популярности (вещественный тип в диапазоне от 0 до 100), наличие абстрактных типов данных (булевский тип, который может принимать значение 1 – true, 0 – false), имя (строка, начинающаяся с заглавной буквы, не превышающая длину 100).
2. *объектно-ориентированный язык программирования*: вид наследования (целочисленный ключ в диапазоне от 1 до 3, где 1 – одиночное наследование, 2 – множественное наследование, 3 – наследование интерфейса), год создания (целочисленный тип в диапазоне от 1940 до 2020), процент популярности (вещественный тип в диапазоне от 0 до 100), имя (строка, начинающаяся с заглавной буквы, не превышающая длину 100).
3. *функциональный язык программирования*: вид типизации (целочисленный ключ в диапазоне от 1 до 2, где 1 – строгая типизация, 2 – динамическая типизация), год создания (целочисленный тип в диапазоне от 1940 до 2020), процент популярности (вещественный тип в диапазоне от 0 до 100), наличие «ленивых» вычислений (булевский тип, который может принимать значение 1 – true, 0 – false), имя (строка, начинающаяся с заглавной буквы, не превышающая длину 100).

В примере (*Изображение 1*) приведен процедурный язык программирования, год создания 2011, процент популярности = 0.294, наличие абстрактных типов данных = true, название – Kotlin.

Все составленные тесты лежат в .txt файлах test1.txt, test2.txt, test3.txt, test4.txt, test5.txt, test6.txt, test7.txt, test8.txt, test9.txt, test10.txt. Файлы для вывода результатов работы программы – result.txt.out и result2.txt.out

**Основные характеристики программы**

Время работы тестовых данных:

1. **Тест 1:**

*2 2 1991 9.7 Python*

*3 2 1958 0.3 1 Lisp*

*1 2011 0.294 1 Kotlin*

Время работы = 0.002s

1. **Тест 2:**

*3 1 2005 0.232 1 F#*

*2 2 1991 9.7 Python*

*2 1 2000 5.3 C#*

*3 2 2004 0.378 1 Scala*

Время работы = 0.002s

1. **Тест 3:**

*1 2009 0.9 1 Go*

*2 1 2000 5.3 C#*

*1 2011 0.294 1 Kotlin*

*3 2 2004 0.378 1 Scala*

Время работы = 0.00197s

1. **Тест 4**

*1*

*2011 0.294 1 Kotlin*

*2*

*1 2000 5.3*

*C#*

Время работы = 0.002s

1. **Тест 5**

*3*

*2 2004 0.378 1 Scala*

*3*

*1 2005 0.232 1 F#*

*2*

*2 1991 9.7 Python*

*3*

*2 1958 0.3 1 Lisp*

*1*

*2011 0.294 1 Kotlin*

Время работы = 0.002s

1. **Тест 6:**

*2*

*2 1991 9.7 Python*

*1*

*1980 0.62 1 Ada*

*3*

*2 1958 0.3 1 Lisp*

*3*

*2 1995 11.68 1 Java*

*1*

*2011 0.294 1 Kotlin*

Время работы = 0.002s

1. **Тест 7:**

*1*

*1980 0.62 1 Ada*

*2*

*1 1995 1.2 Ruby*

*2*

*2 1991 9.7 Python*

*3*

*2 1958 0.3 1 Lisp*

*3*

*2 2004 0.378 1 Scala*

*3*

*1 2005 0.232 1 F#*

Время работы = 0.00203s

1. **Тест 8:**

*3*

*1 2005 0.232 1 F#*

*2*

*1 1995 1.2 Ruby*

*2*

*1 2000 5.3 C#*

*3*

*2 2004 0.378 1 Scala*

*1*

*1980 0.62 1 Ada*

Время работы = 0.002ms

1. **Тест 9:**

*1*

*1980 0.62 1 Ada*

*3*

*2 1995 11.68 1 Java*

*2*

*1 1995 1.2 Ruby*

Время работы = 0.002s

1. **Тест 10:**

*1*

*2020 30.33 1 NewLang2.0*

*3*

*1 1999 1.003 1 Lang2.0*

*3*

*2 1998 2.0333 0 Flang*

*2*

*1 2017 3.8752 OOP-new-Lang*

*1*

*2019 20.22 1 NewLang2.0*

Время работы = 0.00201s

Время работы для рандомного формирования входынх данных в зависимости от их количества:

1: 0.001s

10: 0.002s

100: 0.00503s

500: 0.02904s

1000: 0.08906s

Число интерфейсных модулей = 0.

Число модулей реализации = 6.

Общий размер исходных текстов = 22Кб.

Исполняемый файл не генерируется.

**Сравнительный анализ по полученным характеристикам с реализациями, сделанными в других заданиях**

В текущей работе код реализуется на основе парадигмы динамического программирования.

Формат тестовых входных данных не изменился, но в данном задании вводить первый параметр (и разграничивать базовые альтернативы между собой) можно не только новой строкой, но и через пробел.

Время работы тестовых данных изменилось следующим образом:

|  |  |  |  |
| --- | --- | --- | --- |
| Время работы (в ms) для теста № | Работа 1 | Работа 2 | Работа 3 |
| 1 | 3 | 3 | 2 |
| 2 | 5 | 3 | 2 |
| 3 | 4 | 3 | 1.97 |
| 4 | 6 | 3 | 2 |
| 5 | 18 | 8 | 2 |
| 6 | 4 | 4 | 2 |
| 7 | 6 | 5 | 2.03 |
| 8 | 4 | 4 | 2 |
| 9 | 3 | 3 | 2 |
| 10 | 5 | 4 | 2.01 |

Опираясь на данные составленной таблицы, можно сделать вывод, что программа стала работать быстрее (хотя данная метрика непостоянна и зависит от некоторых внешних факторов, например, тестирующей машины, однако мы можем увидеть явно выраженный результат, который сложно не заметить).

Также было отмечено, что число интерфейсных модулей уменьшилось, а если быть точнее, в данном задании их нет (т.к. задание реализовано в парадигме динамической типизации и данные инициализируются динамически), а вот количество модулей реализации не изменилось. Общий размер исходных текстов уменьшилось на 4Кб, а полученный размер исполняемого кода сравнить не удалось, так как в данном задании исполняемый файл не генерируется. (Все данные можно сравнить в таблице, приведенной ниже)

|  |  |  |
| --- | --- | --- |
| Метрики: | Задание 2 | Задание 3 (текущее) |
| Количество интерфейсных модулей | 6 | 0 |
| Количество исполняемых модулей | 6 | 6 |
| Общий размер исходных текстов | 26Кб | 22Кб |
| Полученный размер исполняемого кода | 110Кб | – |

**Вывод:** данная работа работа имеет преимущество в сравнении с предыдущими в том, что можно использовать классы (в отличие от задания №1) и не писать интерфейсные модули (в отличие от задания №2). Благодаря классам обобщенный артефакт можно вынести в отдельный базовый класс, а базовые альтернативы можно представить с помощью трех классов-наследников, в связи с чем можно явно выделить уникальные параметры каждой альтернативы и показать общие параметры всех альтернатив, оставив их в базовом классе. Можно заметить, что время работы программы по сравнению с предыдущими заданиями уменьшилось, что можно связать с тем, что реализация происходит в парадигме динамического программирования и время на проверку типов перед запуском (хотя данное суждение весьма спорно, так как данная метрика зависит не только от вида типизации, но и от языка программирования, тестирующей машины, компилятора и проч…). Также динамическая типизация дала возможность не писать интерфейсные модули. Общий размер исходных текстов также уменьшился, что можно связать с отсутствием интерфейсных модулей. К недостаткам данной парадигмы можно отнести то, что на этапе тестирования программы приходилось исправлять намного больше ошибок во время исполнения, связанных с типизацией.