Задорожный С.С., Фадеев Е. П.

Объектно-ориентированное программирование на языке Python

Учебно-методическое пособие по дисциплине
«Введение в компьютерные технологии»

Москва
Физический факультет МГУ имени М.В. Ломоносова
2022

Задорожный Сергей Сергеевич, Фадеев Егор Павлович

Объектно-ориентированное программирование на языке Python

М. : Физический факультет МГУ им. М. В. Ломоносова, 2022. – 49 с.

В последнее время языки сценариев, такие как Python, набирают популярность. На них написано больше программного обеспечения, чем на традиционных системных языках. Язык Python доступен для всех основных платформ и активно применяется в научных вычислениях, машинном обучении, веб-разработке, создании игр и визуальных эффектов, и ряде других направлений.

Язык Python позволяет писать программы в традиционном процедурном стиле, однако крупные проекты имеет смысл разрабатывать, используя парадигму объектно-ориентированного программирования (ООП). В языке Python ООП играет ключевую роль, т.к. практически все сущности языка представляют собой объекты.

По сравнению со многими другими языками в Python объектно-ориентированное программирование обладает рядом особых черт, которые следует внимательно рассмотреть при освоении языка. Целью пособия является помощь студентам по освоению этих особенностей.

Рассмотрен ряд программ иллюстрирующих создание собственных классов.

Приведен обширный список литературы (в том числе и интернет ресурсов), который поможет выбрать наиболее подходящее издание в соответствии с пожеланиями и уровнем подготовки.

Пособие поможет студентам более самостоятельно и оперативно решать задачи практикума по программированию.

Рассчитано на студентов первого и второго курсов физического факультета, но может быть полезно студентам старших курсов, аспирантам и сотрудникам, занимающимися разработкой программного обеспечения на языке Python.

Авторы – сотрудники кафедры математического моделирования и информатики
физического факультета МГУ.

Рецензенты: доцент кафедры ОФиВП Коновко А.А., ведущий электроник каф. ОФиВП Лукашев А.А.

Подписано в печать . Объем 3,5 п.л. Тираж 30 экз. Заказ № .

Физический факультет им. М. В. Ломоносова,
119991 Москва, ГСП-1, Ленинские горы, д. 1, стр. 2.

Отпечатано в отделе оперативной печати физического факультета МГУ.

Оглавление

1. Введение

Python (в русском программистском сообществе прижилось название «Питон», но правильнее будет «Пайтон») – это язык с динамической типизацией данных, который поддерживает многие парадигмы программирования: процедурное, ООП, функциональное.

Достоинства языка:

- Удобный синтаксис.
- Скорость разработки.
- Большое количество "сахара" от разработчиков языка.
- Кроссплатформенность (достаточно наличия необходимых библиотек и интерпретатора, чтобы запустить вашу программу на другой ОС).
- Большое количество библиотек позволяет не изобретать велосипеды.
- Широко используется (компания Google использует Python в своей поисковой системе, компании Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, IBM используют Python для тестирования аппаратного обеспечения, компании JPMorgan Chase, UBS, Getco и Citadel применяют Python для прогнозирования финансового рынка, , служба коллективного использования видеоматериалов YouTube и популярная программа BitTorrent для обмена файлами в пиринговых сетях тоже написаны на языке Python, NASA, Los Alamos, JPL и Fermilab используют Python для научных вычислений).

Циклы, ветвления и функции – все это элементы структурного программирования. Его возможностей вполне хватает для написания небольших, простых программ и сценариев. Однако крупные проекты часто реализуют, используя парадигму объектно-ориентированного программирования (ООП). В языке Python ООП играет ключевую роль. Даже программируя в рамках структурной модели, вы все равно пользуетесь объектами и классами, пусть даже встроенными в язык, а не созданными лично вами.

В Python все объекты являются производными классов и наследуют от них атрибуты. При этом каждый объект формирует собственное пространство имен. Python поддерживает такие ключевые особенности объектно-ориентированного программирования как наследование, инкапсуляцию и полиморфизм.

Полиморфизм позволяет объектам разных классов иметь схожие интерфейсы. Он реализуется путем объявления в них методов с одинаковыми именами. К проявлению полиморфизма как особенности ООП также можно отнести методы перегрузки операторов.

Кроме наследования, инкапсуляции и полиморфизма существуют другие особенности ООП. Таковой является композиция, или агрегирование, когда класс включает в себя вызовы других классов. В результате при создании объекта от класса-агрегата, создаются объекты других классов, являющиеся составными частями первого.

Преимущества ООП

Особенности объектно-ориентированного программирования наделяют его рядом преимуществ.

Так ООП позволяет использовать один и тот же программный код с разными данными. На основе классов создается множество объектов, у каждого из которых могут быть собственные значения полей. Нет необходимости вводить множество переменных, т.к. объекты получают в свое распоряжение индивидуальные пространства имен. В этом смысле объекты похожи на структуры данных. Объект можно представить в виде некой упаковки данных, к которой присоединены инструменты для их обработки – методы.

Наследование позволяет не писать новый код, а использовать и настраивать уже существующий за счет добавления и переопределения атрибутов.

Недостатки ООП

ООП позволяет оптимизировать дальнейшую поддержку разрабатываемого приложения, однако предполагает большую роль предварительного анализа предметной области и проектирования. От правильности решений на этом этапе зависит куда больше, чем от непосредственного написания исходного кода.

Следует понимать, что одна и та же задача может быть решена разными объектными моделями, каждая из которых будет иметь свои преимущества и недостатки. Только опытный разработчик может сказать, какую из них будет проще расширять и обслуживать в дальнейшем.

Особенности ООП в Python

По сравнению со многими другими языками в Python объектно-ориентированное программирования обладает рядом особых черт.

Всё является объектом – число, строка, список, функция, экземпляр класса, сам класс, модуль. Так класс – объект, способный порождать другие объекты – экземпляры.

В Python нет просто типов данных. Все типы – это классы.

Инкапсуляции в Python не уделяется особого внимания. В других языках программирования обычно нельзя получить напрямую доступ к свойству, описанному в классе. Для его изменения может быть предусмотрен специальный метод. В Python же не считается предосудительным непосредственное обращение к свойствам.

2. Типы данных. Переменные.

В реальной жизни мы совершаем различные действия над окружающими нас предметами, или объектами. Мы меняем их свойства, наделяем новыми функциями. По аналогии с этим компьютерные программы также управляют объектами, только виртуальными, цифровыми. Будем называть такие объекты данными.

Очевидно, данные бывают разными. Часто компьютерной программе приходится работать с числами и строками. Числа в свою очередь также бывают разными: целыми, вещественными, могут иметь огромное значение или очень длинную дробную часть.

При знакомстве с языком программирования Python мы столкнемся с тремя типами данных:

- целые числа (тип *int*) – положительные и отрицательные целые числа, а также 0 (например, 4, 687, -45, 0).
- числа с плавающей точкой (тип *float*) – дробные, они же вещественные, числа (например, 1.45, -3.789654, 0.00453).
- строки (тип *str*) — набор символов, заключенных в кавычки (например, *"ball"*, *"What is your name?"*, *'dkfjUUv'*, *'6589'*). Кавычки в Python могут быть одинарными или двойными. Одиночный символ в кавычках также является строкой, отдельного символьного типа в Python нет.

Данные хранятся в памяти компьютера. Когда мы вводим число, оно помещается в какую-то ячейку памяти. Но как потом узнать, куда именно? Механизм связи между переменными и данными может различаться в зависимости от языка программирования и типов данных. Для этого данные связываются с каким-либо именем и в дальнейшем обращение к ним возможно по этому имени – переменной.

Слово "переменная" обозначает, что сущность может меняться, она непостоянна.

Одна и та же переменная может быть связана сначала с одними данными, а потом – с другими. То есть ее значение может меняться, она *переменчива*.

В программе на языке Python, как и на большинстве других языков, связь между данными и переменными устанавливается с помощью знака =. Такая операция называется

присваивание. Например, выражение *sq = 4* означает, что на объект, представляющий собой число 4, находящееся в определенной области памяти, теперь **ссылается** переменная sq, и обращаться к этому объекту следует по имени sq. Обратите внимание на эту особенность переменных в Python. Если теперь написать *sq=1*, то sq будет ссылаться на **другой объект**, представляющий собой число 1. Т.е. все переменные в Python являются ссылками, переменных закрепленных за фиксированным адресом памяти (как в языке C) нет, соответственно нет и указателей.

3. Вывод данных. Функция print()

Для вывода данных на экран используется функция *print()*.

Можно передавать в функцию *print()* как непосредственно литералы, так и переменные, вместо которых будут выведены их значения. Аргументы функции разделяются между собой запятыми. В выводе вместо запятых значения разделены пробелом. Если в скобках стоит выражение, то сначала оно выполняется, после чего *print()* уже выводит результат данного выражения:

```
print("hello" + " " + "world") # вывод: hello world
print(10 - 2.5/2)   # вывод:   8.75
```

В функции предусмотрены дополнительные параметры. Например, через параметр *sep* можно указать отличный от пробела разделитель строк:

```
print("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun", sep="-")
#вывод: Mon-Tue-Wed-Thu-Fri-Sat-Sun
print(1, 2, 3, sep="//") #вывод: 1//2//3
```

Параметр *end* позволяет указывать, что делать, после вывода строки. По умолчанию происходит переход на новую строку. Однако это действие можно отменить, указав любой другой символ или строку:

```
print(10, end=":") # вывод: 10:
```

В функцию *print()* нередко передаются так называемые форматированные строки, хотя по смыслу их правильнее называть строки-шаблоны. Никакого отношения к самой *print()*они не имеют. Когда такая строка находится в скобках *print()*, интерпретатор сначала согласно заданному в ней формату преобразует ее к обычной строке, после чего передает результат в *print()*.

Форматирование может выполняться в так называемом старом стиле или с помощью строкового метода *format*. Старый стиль также называют Си-стилем, так как он схож с тем, как происходит вывод на экран в языке C:

```
pupil = "Ben"
old = 16
grade = 9.2
print("It's %s, %d. Level: %.2f" % (pupil, old, grade))
# вывод: It's Ben, 16. Level: 9.20
```

Здесь вместо трех комбинаций символов *%s, %d, %f* подставляются значения переменных *pupil, old, grade*. Буквы *s, d, f* обозначают типы данных – строку, целое число, вещественное число.

Теперь посмотрим на метод *format()*:

```
print("This is a {0}. It's {1}.".format("ball", "red"))
# вывод: This is a ball. It's red.
print("This is a {0}. It's {1}.".format("cat", "white"))
# вывод: This is a cat. It's white.
print("This is a {0}. It's {1} {2}.".format(1, "a", "number"))
# вывод: This is a 1. It's a number.
```

В строке в фигурных скобках указаны номера данных, которые будут сюда подставлены. Далее к строке применяется метод *format()*. В его скобках указываются сами данные (можно использовать переменные). На нулевое место подставится первый аргумент метода *format()*, на место с номером 1 – второй и т. д.

В новых релизах Питона появился третий способ создания форматированных строк – *f*-строки. Перед их открывающей кавычкой прописывается буква *f*. В самой строке внутри фигурных скобок записываются выражения на Python, которые исполняются, когда интерпретатор преобразует строку-шаблон в обычную.

```
a = 10
b = 1.33
c = 'Box'
print(f'qty - {a:5}, goods - {c}')
# вывод: qty -    10, goods - Box
print(f'price - {b + 0.2:.1f}')
# вывод: price - 1.5
```

В примере число 5 после переменной a обозначает количество знакомест, отводимых под вывод значения переменной. В выражении *b + 0.2:.1f* сначала выполняется сложение, после этого значение округляется до одного знака после запятой.

4. Ввод данных. Функция input()

За ввод в программу данных с клавиатуры в Python отвечает функция input() . Когда вызывается эта функция, программа останавливает свое выполнение и ждет, когда пользователь введет текст. После того, как он нажмет клавишу Enter, функция input() заберет введенный текст и передаст его программе. Их можно присвоить переменной.

```
town=input()
user=input()
print(f'Вас зовут {user}. Ваш город {town}')
```

Чтобы не вводить человека в замешательство, для функции input() предусмотрен специальный параметр-приглашение. Это приглашение выводится на экран при вызове input():

```
town=input("Ваш город:")
user=input("Ваше имя:")
print(f'Вас зовут {user}. Ваш город {town}')
```

Обратите внимание, что в программу поступает строка. Даже если ввести число, функция input() все равно вернет его строковое представление. Чтобы получить число, надо использовать функции преобразования типов.

```
qty=int(input("Количество:"))
price=float(input("Цена:"))
print(f'Заплатите {qty*price} рублей')
```

В данном случае с помощью функций int() и float() строковые значения переменных qty и price преобразуются соответственно в целое число и вещественное число.

5. Логические выражения и условные операторы

Ранее мы познакомились с тремя типами данных – целыми и вещественными числами, а также строками. Введем четвертый – логический тип данных (тип bool). У этого типа всего два возможных значения: True и False.

В программировании False обычно приравнивают к нулю, а True – к единице. Здесь также работает правило: всё, что не 0 и не пустота, является правдой.

Говоря на естественном языке (например, русском) мы обозначаем сравнения словами "равно", "больше", "меньше". В языках программирования используются специальные знаки, подобные тем, которые используются в математике: > (больше), < (меньше), >= (больше или равно), <= (меньше или равно), == (равно), != (не равно).

Не путайте операцию присваивания значения переменной, обозначаемую в языке Python одиночным знаком "равно", и операцию сравнения (два знака "равно"). Присваивание и сравнение – разные операции.

```
a = 10
b = 5
c = a+b > 14
print(c)   # вывод: True
c = a < 14-b
print(c)   # вывод: False
c = a <= b+5
print(c)   # вывод: True
c = a != b
print(c)   # вывод: True
c = a == b
print(a, b, c) # вывод: 10 5 False
```

Логические выражения типа *kByte >= 1023* являются простыми, так как в них выполняется только одна логическая операция. Однако, на практике нередко возникает необходимость в более сложных выражениях. Может понадобиться получить ответа "Да" или "Нет" в зависимости от результата выполнения нескольких простых выражений. В таких случаях используются специальные операторы, объединяющие два и более простых логических выражения. Широко используются два оператора – так называемые логические *И(and) и ИЛИ (or).*

Чтобы получить *True* при использовании оператора *and*, необходимо, чтобы результаты обоих простых выражений, которые связывает данный оператор, были истинными. Если хотя бы в одном случае результатом будет *False*, то и все выражение будет ложным.

```
x = 8
y = 13
print(y < 15 and x > 8) # вывод: False
```

Чтобы получить *True* при использовании оператора or, необходимо, чтобы результат хотя бы одного простого выражения, входящего в состав сложного, был истинным. В случае оператора or выражение становится ложным лишь тогда, когда ложны оба составляющие его простые выражения.

```
x = 8
y = 13
print(y < 15 or x > 8) # вывод: True
```

В языке Python есть еще унарный логический оператор *not* , то есть отрицание. Он превращает правду в ложь, а ложь в правду. Унарный он потому, что применяется к одному выражению, стоящему после него.

```
print(not y < 15) # вывод: False
```

При выполнении кода, в зависимости от тех или иных условий, некоторые его участки могут быть опущены, в то время как другие – выполнены. Иными словами, в программе может присутствовать ветвление, которое реализуется условным оператором, который имеет следующий формат:

```
if <Логическое выражение> :
      <Блок,  выполняемый,  если условие истинно>
[elif <Логическое выражение> :
     <Блок,  выполняемый,  если условие истинно>
]
[else :
   <Блок,  выполняемый,  если все условия ложны>
]
```

Python считается языком с ясным синтаксисом и легко читаемым кодом. Это достигается сведением к минимуму таких вспомогательных элементов как скобок и точек

с запятой. Для разделения выражений используется переход на новую строку, а для обозначения вложенных выражений – отступы от начала строки:

```
a = 50
n = 98
if n < 100 :
   b = n + a #блок выполняемый, если условие истинно
else :
   b=0       #блок выполняемый, если условие ложно
print(b) #вывод 148
```

Последняя строчка кода print(b) уже не относится к условному оператору, что обозначено отсутствием перед ней отступа.

В язык Python встроена возможность настоящего множественного ветвления на одном уровне вложенности, которое реализуется с помощью веток elif. Слово "elif" образовано от двух первых букв слова "else", к которым присоединено слово "if". Это можно перевести как "иначе если". В отличие от else , в заголовке elif обязательно должно быть логическое выражение также, как в заголовке if:

```
old = int(input('Ваш возраст: '))
print('Рекомендовано:', end=' ')
if old < 6:
    print('"Заяц в лабиринте"')
elif 6 <= old < 12:
    print('"Марсианин"')
elif 12 <= old < 16:
    print('"Загадочный остров"')
else:
    print('"Поток сознания"')
```

6. Списки

Список в Python – это встроенный тип (класс) данных, представляющий собой одну из разновидностей структур данных. Структуру данных можно представить, как сложную единицу, объединяющую в себе группу более простых. Каждая разновидность структур данных имеет свои особенности. Список – это изменяемая последовательность произвольных элементов.

В большинстве других языков программирования есть такой широко используемый тип данных как массив. В Питоне такого встроенного типа нет. Однако списки условно можно считать аналогом массивов за одним исключением. Составляющие массив элементы должны принадлежать одному типу данных, для списков такого ограничения нет.

Например, массив может содержать только целые числа или только вещественные числа или только строки. Список также может содержать элементы только одного типа, что делает его внешне неотличимым от массива. Но вполне допустимо, чтобы в одном списке содержались как числа, так и строки, а также что-нибудь еще.

Создавать списки можно разными способами. Создадим его простым перечислением элементов:

```
a = [12, 3.85, "black", -4]
```

Итак, у нас имеется список, присвоенный переменной a. В Python список определяется квадратными скобками. Он содержит четыре элемента. Если где-то в программе нам понадобится весь этот список, мы получим доступ к нему, указав всего лишь одну переменную – a.

Элементы в списке упорядочены, имеет значение в каком порядке они расположены. Каждый элемент имеет свой индекс, или номер. Индексация начинается с нуля. В данном

случае число 12 имеет индекс 0, строка "black" – индекс 2. Чтобы извлечь конкретный элемент, надо после имени переменной указать в квадратных скобках его индекс:

```
print(a[0])
```

В Python существует также индексация с конца. Она начинается с -1:

```
print(a[-1])
```

Часто требуется извлечь не один элемент, а так называемый срез – часть списка. В этом случае указывается индекс первого элемента среза и индекс следующего за последним элементом среза:

```
print(a[0:2]) # [12, 3.85]
```

В данном случае извлекаются первые два элемента с индексами 0 и 1. Элемент с индексом 2 в срез уже не входит. В таком случае возникает вопрос, как извлечь срез, включающий в себя последний элемент? Если какой-либо индекс не указан, то считается, что имеется в виду начало или конец:

```
print(a[:3]) # [12, 3.85, 'black']
print(a[2:]) # ['black',-4]
print(a[:])  # [12, 3.85, 'black',-4]
```

Списки – изменяемые объекты. Это значит, что в них можно добавлять элементы, удалять их, изменять существующие. Проще всего изменить значение элемента. Для этого надо обратиться к нему по индексу и перезаписать значение в заданной позиции:

```
a[1] = 4
```

Добавлять и удалять лучше с помощью специальных встроенных методов списка:

```
a.append('wood')
a.insert(1, 'circle')
a.remove(4)
a.pop()
a.pop(2)
```

Перечень всех методов списка можно посмотреть в интернете, например, на официальном сайте https://docs.python.org/3/tutorial/datastructures.html

Можно изменять списки не используя методы, а с помощью взятия и объединения срезов:

```
b = [1, 2, 3, 4, 5, 6]
b = b[:2] + b[3:]
```

Здесь берется срез из первых двух элементов и срез, начиная с четвертого элемента (индекс 3) и до конца. После чего срезы объединяются с помощью оператора "сложения".

Можно изменить не один элемент, а целый срез:

```
mylist = ['ab','ra','ka','da','bra']
mylist[0:2] = [10,20]
print(mylist) #  [10, 20, 'ka', 'da', 'bra']
```

7. Циклы

Циклы являются такой же важной частью структурного программирования, как условные операторы. С помощью циклов можно организовать повторение выполнения участков кода.

Язык Python при помощи циклов позволяет компактно записать многие повторяющиеся действия, например, вывод чисел от 1 до 100:

```
for x in range (1, 101):  print(x)
```

Цикл for применяется для перебора элементов последовательности и имеет такой формат:

```
for <Текущий элемент> in <Последовательность> :
    <Инструкции внутри цикла>
[else:
    <Блок,  выполняемый,  если не использовался оператор break>
]
```

Здесь присутствуют следующие конструкции:

- <Последовательность> – объект, поддерживающий механизм итерации (последовательного перебора): строка, список, кортеж, диапазон, словарь и др.
- <Текущий элемент> – на каждой итерации через эту переменную доступен очередной элемент последовательности или ключ словаря.
- <Инструкции внутри цикла> – блок, который будет многократно выполняться.
- Если внутри цикла не использовался оператор break, то после завершения выполнения цикла будет выполнен блок в инструкции else. Этот блок не является обязательным.

Пример перебора букв в слове:

```
for symbol in "string":
   print(symbol,  end="  ")
else:
   print("\nЦикл выполнен")
```

Функцией range () используется для генерации индексов. Она имеет следующий формат: *range ([<Начало>,]<Конец>[, <Шаг>])*

Первый параметр задает начальное значение. Если параметр <Начало> не указан, то по умолчанию используется значение 0. Во втором параметре указывается конечное значение. Следует заметить, что оно не входит в возвращаемые значения. Если параметр <Шаг> не указан, то используется значение 1. Функция возвращает диапазон – особый объект, поддерживающий итерационный протокол. С помощью диапазона внутри цикла for можно получить значение текущего элемента. В качестве примера умножим каждый элемент списка на 2:

```
arr =  [1,  2,  3]
for i  in range (len (arr) ) :
   arr[i]  *= 2
print (arr) # вывод [2,4,6]
```

Цикл while

Выполнение инструкций в цикле while продолжается до тех пор, пока логическое выражение истинно. Он имеет следующий формат:

```
while <Условие> :
    <Инструкции>
[else:
    <Блок, выполняемый, если не использовался оператор break>
]
```

Последовательность работы цикла *while* следующая. Инструкции внутри цикла выполняются пока условие истинно. При этом истинность условия проверяется перед каждой итерацией цикла. Выход из цикла осуществляется либо если условие не прошло очередную проверку на истинность, что может случиться и на входе в цикл, либо если встречается оператор *break.*

Опционально можно дополнить цикл *while,* блоком *else,* инструкции в котором выполнятся по завершении цикла, при условии, что не сработал оператор *break.*

В отличии от цикла *for,* единственная возможность которого пробежаться по итерируемому объекту, цикл *while* позволяет писать циклы, выход из которых осуществляется при более сложных обстоятельствах.

В качестве примера, рассмотрим поиск наибольшего общего знаменателя методом Эвклида, итерации которого выполняются до тех пор, пока наименьшее из чисел не равно нулю.

```
x, y = 60, 48
while y:
    x, y = y, x % y
print(x) # вывод 12
```

Нередко цикл *while* комбинируют с счетчиком. Тогда он принимает следующий формат:

```
<Задание начального значения для переменной - счетчика>
while <Условие> :
    <Инструкции>
    <Приращение значения в переменной - счетчике>
[else:
    <Блок, выполняемый, если не использовался оператор break>
]
```

Последовательность его работы сводится к:

1. Переменной-счетчику присваивается начальное значение.
2. Проверяется условие, если оно истинно, то выполняются инструкции внутри цикла, иначе выполнение цикла завершается.
3. Переменная-счетчик изменяется на величину, указанную в параметре <приращение>.
4. Переход к пункту 2.
5. Если внутри цикла не использовался оператор *break*, то после завершения выполнения цикла будет выполнен блок в инструкции *else* . Этот блок не является обязательным.

Создадим пустой список и заполним его в цикле случайными числами:

```
import random # модуль генерации случайных чисел
c = []
i = 0
while i < 10:
    c.append(random.randint(0,100))
    i += 1
print(c)
```

Оператор *continue* позволяет перейти к следующей итерации цикла до завершения выполнения всех инструкций внутри цикла. В качестве примера выведем все числа от 1 до 100, кроме чисел от 5 до 10 включительно:

```
for i in range(1,101):
   if 4 < i < 11 :
      continue  # Переходим на следующую итерацию цикла
   print(i)
```

Оператор *break* позволяет прервать выполнение цикла досрочно. Для примера выведем все числа от 1 до 100 еще одним способом:

```
i = 1
while True:
   if i > 100:  break  # Прерываем цикл
   print(i)
   i += 1
```

Цикл *while* совместно с оператором *break* удобно использовать для получения неопределенного заранее количества данных от пользователя. В качестве примера просуммируем произвольное количество чисел:

```
print("Введите слово  'stop'  для получения результата")
summa = 0
while True:
   x = input("Введите число:  ")
   if x == "stop":
      break  # Выход из цикла
   x = int(x)  # Преобразуем строку в число
   summa += x
print("Сумма чисел равна:",  summa)
```

8. Функции

Функция в программировании представляет собой обособленный участок кода, который можно вызывать, обратившись к нему по имени, которым он был назван. При вызове происходит выполнение команд тела функции.

Функции можно сравнить с небольшими программками, которые сами по себе, то есть автономно, не исполняются, а встраиваются в обычную программу. Нередко их так и называют – подпрограммы. Других ключевых отличий функций от программ нет. Функции также при необходимости могут получать и возвращать данные. Только обычно они их получают не с ввода (клавиатуры, файла и др.), а из вызывающей программы. Сюда же они возвращают результат своей работы.

Существует множество встроенных в язык программирования функций. С некоторыми такими в Python мы уже сталкивались. Это *print(), input(), int(), float(), str(), type()*. Код их тела нам не виден, он где-то "спрятан внутри языка". Нам же предоставляется только интерфейс – имя функции.

С другой стороны, программист всегда может определять свои функции. Их называют пользовательскими.

В языке программирования Python функции определяются с помощью оператора *def* .

Рассмотрим код:

```
def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")
```

Это пример определения функции. Как и другие сложные инструкции вроде условного оператора и циклов функция состоит из заголовка и тела. Заголовок оканчивается двоеточием и переходом на новую строку. Тело имеет отступ.

Ключевое слово *def* сообщает интерпретатору, что перед ним определение функции. За d*ef* следует имя функции. Оно может быть любым, так же, как и всякий идентификатор. В программировании весьма желательно давать всему осмысленные имена. Так в данном случае функция названа "посчитать_еду" в переводе на русский. После имени функции ставятся скобки. В приведенном примере они пустые. Это значит, что функция не принимает никакие данные из вызывающей ее программы. Однако она могла бы их принимать, и тогда в скобках были бы указаны параметры этой функции.

После двоеточия следует тело, содержащее инструкции, которые выполняются при вызове функции. Следует различать определение функции и ее вызов. В программном коде они не рядом и не вместе. Можно определить функцию, но ни разу ее не вызвать. Нельзя вызвать функцию, которая не была определена. Определив функцию, но ни разу не вызвав ее, вы никогда не выполните ее тела.

```
def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")
print("Сколько бананов и ананасов для обезьян?")
count_food()
print("Сколько жуков и червей для ежей?")
count_food()
print("Сколько рыб и моллюсков для выдр?")
count_food()
```

В языке Python определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строка за строкой и о том, что находится ниже по течению, ему еще неизвестно. Поэтому если вызов функции предшествует ее определению, то возникает ошибка (выбрасывается исключение NameError).

Польза функций не только в возможности многократного вызова одного и того же кода из разных мест программы. Не менее важно, что благодаря им программа обретает истинную структуру. Функции как бы разделяют ее на обособленные части, каждая из которых выполняет свою конкретную задачу.

Функции могут передавать какие-либо данные из своих тел в основную ветку программы. Говорят, что функция возвращает значение. В большинстве языков программирования, в том числе Python, выход из функции и передача данных в то место, откуда она была вызвана, выполняется оператором *return* .

Если интерпретатор Питона, выполняя тело функции, встречает *return* , то он "забирает" значение, указанное после этой команды, и "уходит" из функции.

```
def cylinder():
    r = float(input())
    h = float(input())
    # площадь боковой поверхности цилиндра:
    side = 2 * 3.14 * r * h
    # площадь одного основания цилиндра:
    circle = 3.14 * r**2
    # полная площадь цилиндра:
    full = side + 2 * circle
    return full
square = cylinder()
print(square)
```

В Питоне позволительно возвращать из функции несколько объектов, перечислив их через запятую после команды *return*:

```
def cylinder():
    r = float(input())
    h = float(input())
    side = 2 * 3.14 * r * h
    circle = 3.14 * r**2
    full = side + 2 * circle
    return side, full
sCyl, fCyl = cylinder()
print("Площадь боковой поверхности %.2f" % sCyl)
print("Полная площадь %.2f" % fCyl)
```

Из функции *cylinder()* возвращаются два значения. Первое из них присваивается переменной *sCyl*, второе – *fCyl*.

Фокус здесь в том, что перечисление значений через запятую (например, 10, 15, 19) создает объект типа *tuple*. На русский переводится как "кортеж". Когда же кортеж присваивается сразу нескольким переменным, то происходит сопоставление его элементов соответствующим в очереди переменным. Это называется распаковкой.

Таким образом, когда из функции возвращается несколько значений, на самом деле из нее возвращается один объект класса *tuple*. Перед возвратом эти несколько значений упаковываются в кортеж. Если же после оператора *return* стоит только одна переменная или объект, то ее/его тип сохраняется как есть.

Функции могут не только возвращать данные, но также принимать их, что реализуется с помощью так называемых параметров, которые указываются в скобках в заголовке функции. Количество параметров может быть любым.

Параметры представляют собой локальные переменные, которым присваиваются значения в момент вызова функции. Конкретные значения, которые передаются в функцию при ее вызове, называются аргументами.

Когда функция вызывается, то ей передаются фактические значения аргументов. Т.е. переменным-параметрам присваиваются переданные в функцию значения. Соответственно, изменение значений в теле функции никак не скажется на значениях фактических переменных. Они останутся прежними. В Python такое поведение характерно для неизменяемых типов данных, к которым относятся, например, числа и строки.

Существуют изменяемые типы данных. Для Питона, это, например, списки и словари. В этом случае данные передаются по ссылке. В функцию передается ссылка на них, а не сами данные. И эта ссылка связывается с локальной переменной. Изменения таких данных через локальную переменную обнаруживаются при обращении к ним через глобальную. Это есть следствие того, что несколько переменных ссылаются на одни и те же данные, на одну и ту же область памяти. Необходимость передачи по ссылке связана в первую очередь с экономией памяти. Сложные типы данных, по сути представляющие собой структуры данных, обычно копировать не целесообразно. Однако, если надо, всегда можно сделать это принудительно.

9. Модули

Встроенные в язык программирования функции доступны сразу. Чтобы их вызвать, не надо выполнять никаких дополнительных действий. Однако за время существования языка на нем было написано столько функций и классов, которые оказались востребованными множеством программистов и в разных областях, что включить весь этот объем кода в сам язык если возможно, то нецелесообразно.

Чтобы разрешить проблему доступа к дополнительным возможностям языка, в программировании стало общепринятой практикой использовать так называемые модули, пакеты и библиотеки. Каждый модуль содержит коллекцию функций и классов, предназначенных для решения задач из определенной области. Так в модуле math языка Python содержатся математические функции, модуль random позволяет генерировать псевдослучайные числа, в модуле datetime содержатся классы для работы с датами и временем, модуль sys предоставляет доступ к системным переменным и т. д.

Количество модулей для языка Python огромно, что связано с популярностью языка. Часть модулей собрана в так называемую стандартную библиотеку. Стандартная она потому, что поставляется вместе с установочным пакетом. Однако существуют сторонние библиотеки. Они скачиваются и устанавливаются отдельно.

Для доступа к функционалу модуля, его надо импортировать в программу. После импорта интерпретатор "знает" о существовании дополнительных классов и функций и позволяет ими пользоваться.

В Питоне импорт осуществляется командой import . При этом существует несколько способов импорта. Рассмотрим работу с модулем на примере math:

```
import math
```

В глобальной области видимости появилось имя math . Если бы до импорта вы упомянули бы имя math , то возникла бы ошибка NameError . Теперь же в программе завелся объект math , относящийся к классу module .

Чтобы вызвать функцию из модуля, надо впереди написать имя модуля, поставить точку, далее указать имя функции, после чего в скобках передать аргументы, если они требуются. Например, чтобы вызвать функцию pow из math , надо написать так:

```
print(math.pow(2, 2)) # 4.0
```

Для обращения к константе скобки не нужны:

```
print(math.pi) # 3.141592653589793
```

Описание модулей и их содержания можно посмотреть в официальной документации на сайте python.org.

Второй способ импорта позволяет импортировать не весь модуль, а только необходимые функции из него:

```
from math import gcd, sqrt, hypot
```

Перевести можно как "из модуля math импортировать функции gcd , sqrt и hypot ".

В таком случае при их вызове не надо перед именем функции указывать имя модуля:

```
from math import gcd, sqrt, hypot
print(gcd(100, 150)) # 50
print(sqrt(16))      # 4.0
print(hypot(3, 4))   # 5.0
```

Чтобы импортировать сразу все функции из модуля:

```
from math import *
```

Импорт через from не лишен недостатка. В программе уже может быть идентификатор с таким же именем, как имя одной из импортируемых функций или констант. Ошибки не будет, но одно из них окажется "затерто":

```
pi = 3.14
print(pi) # 3.14
from math import pi
print(pi)  # 3.141592653589793
```

Здесь исчезает значение 3.14, присвоенное переменной pi. Это имя теперь указывает на число из модуля math.

В этой связи более опасен именно импорт всех функций. Так как в этом случае очень легко не заметить подмены значений идентификаторов. Однако можно изменить имя идентификатора из модуля на какое угодно:

```
pi = 3.14
from math import pi as P
print(P)  # 3.141592653589793
print(pi) # 3.14
```

В данном случае константа pi из модуля импортируется под именем P. Смысл подобных импортов в сокращении имен, так как есть модули с длинными именами, а имена функций и классов в них еще длиннее. Если в программу импортируется всего пара сущностей, и они используются в ней часто, то имеет смысл переименовать их на более короткий вариант.

Сравните:

```
import calendar
print(calendar.weekheader(2)) # Mo Tu We Th Fr Sa Su
```

и

```
from calendar import weekheader as week
print(week(3))  # Mon Tue Wed Thu Fri Sat Sun
```

Во всех остальных случаях лучше оставлять идентификаторы содержимого модуля в пространстве имен самого модуля и получать доступ к ним через имя модуля, то есть выполнять импорт командой *import имя_модуля*, а вызывать, например, функции через *имя_модуля.имя_функции()*.

10. Объектно-ориентированное программирование

Итак, что же такое объектно-ориентированное программирование? Судя по названию, ключевую роль здесь играют некие объекты, на которые ориентируется весь процесс программирования.

Реальный мир для нас представляется в виде множества объектов, обладающих определенными свойствами, взаимодействующих между собой и вследствие этого изменяющимися. Эта привычная для взгляда человека картина мира была перенесена в программирование.

Ключевую разницу между программой, написанной в структурном стиле, и объектно-ориентированной можно выразить так. В первом случае, на первый план выходит логика, понимание последовательности выполнения действий для достижения поставленной цели. Во втором – важнее представить программу как систему объектов, взаимодействие которых способно решить ту или иную задачу.

Объектно-ориентированное программирование – это способ организации программы, позволяющий использовать один и тот же код многократно. В отличие от функций и модулей, ООП позволяет не только разделить программу на фрагменты, но и описать предметы реального мира в виде удобных сущностей — объектов, а также организовать связи между этими объектами.

Основным «кирпичиком» ООП является класс — сложный тип данных, включающий набор переменных и функций для управления значениями, хранящимися в этих переменных.

Переменные называют атрибутами или свойствами, а функции — методами. Класс является фабрикой объектов, т. е. позволяет создать неограниченное количество экземпляров, основанных на этом классе.

Основными понятиями, используемыми в ООП, являются класс, объект, наследование, инкапсуляция и полиморфизм. В языке Python класс равносилен понятию тип данных.

Что такое **класс** или тип? Проведем аналогию с реальным миром. Если мы возьмем конкретный стол, то это **объект**, но не класс. А вот общее представление о столах, их назначении – это класс. Ему принадлежат все реальные объекты столов, какими бы они ни были. Класс столов дает общую характеристику всем столам в мире, он их обобщает.

То же самое с целыми числами в Python. Тип *int* – это класс целых чисел. Числа 5, 100134, -10 и т. д. – это конкретные объекты этого класса.

В языке программирования Python объекты класса принято называть также **экземплярами**. Это связано с тем, что в нем все классы сами являются объектами класса *type*. Точно также как все модули являются объектами класса *module*.

Следующее по важности понятие объектно-ориентированного программирования – **наследование**. Вернемся к столам. Пусть есть класс столов, описывающий общие свойства всех столов. Однако можно разделить все столы на письменные, обеденные и журнальные и для каждой группы создать свой класс, который будет наследником общего класса, но также вносить ряд своих особенностей. Таким образом, общий класс будет родительским, а классы групп – дочерними, производными.

Дочерние классы наследуют особенности родительских, однако дополняют или в определенной степени модифицируют их характеристики. Когда мы создаем конкретный

экземпляр стола, то должны выбрать, какому классу столов он будет принадлежать. Если он принадлежит классу журнальных столов, то получит все характеристики общего класса столов и класса журнальных столов. Но не особенности письменных и обеденных.

Основное (но не единственное) преимущество, которое дает концепция наследования в программировании, – это вынос одинакового кода из разных классов в один родительский класс. Другими словами, наследование позволяет сводить на нет повторение кода в разных частях программы.

Инкапсуляция в ООП понимается двояко. Во многих языках этот термин обозначает сокрытие данных, то есть невозможность напрямую получить доступ к внутренней структуре объекта, так как это небезопасно. Например, наполнить желудок едой можно напрямую, положив еду в желудок. Но это опасно. Поэтому прямой доступ к желудку закрыт. Чтобы наполнить его едой, надо совершить ритуал, через элемент интерфейса под названием рот.

В Python нет такой инкапсуляции, хотя она является одним из стандартов ООП. В Python можно получить доступ к любому атрибуту объекта и изменить его. Однако есть механизм, позволяющий имитировать сокрытие данных, если это так уж необходимо.

Отсутствие сокрытия данных в Python делает программирование на нем проще, но привносит ряд особенностей, связанных с пространствами имен.

Второй смысл инкапсуляции – объединение описания свойств объектов и их поведения в единое целое, то есть в класс. Инкапсуляция в этом смысле вытекает из самой идеи объектно-ориентированного программирования и, соответственно, имеется во всех ОО-языках.

Полиморфизм можно перевести как множество форм. В ООП под полиморфизмом понимается следующее. Объекты разных классов, с разной внутренней реализацией, то есть программным кодом, могут иметь "одинаковые" методы. На самом деле у методов совпадают только имена, а вложенный в них код (то, что они делают) различен. Вот и получается, что у одного имени как бы множество форм.

Например, для чисел есть операция сложения, обозначаемая знаком +. Однако мы можем определить класс, объекты которого также будут поддерживать операцию, обозначаемую этим знаком. Но это вовсе не значит, что объекты должны быть числами, и будет получаться какая-то сумма. Операция + для объектов нашего класса может значить что-то иное. Но интерфейс, в данном случае это знак +, у чисел и нашего класса будет одинаков. А полиморфизм проявляется во внутренней реализации и результате операции.

Полиморфизм полезен не только тем, что дает возможность объектам пользовательских классов участвовать в стандартных операциях. Если у объектов разных классов есть одноименный метод, то коллекция таких разнородных объектов может быть обработана в одном цикле.

11. Создание классов и объектов

Класс описывается с помощью ключевого слова *class* по следующей схеме:

```
class <Название класса>[(<Класс1>, ..., <Класс№>)]:
    """ Строка документирования """
    <Описание атрибутов и методов>
```

Инструкция создает новый объект и присваивает ссылку на него идентификатору, указанному после ключевого слова class. Это означает, что название класса должно полностью соответствовать правилам именования переменных. После названия класса в круглых скобках можно указать один или несколько базовых классов через запятую. Если же класс не наследует базовые классы, то круглые скобки можно не указывать. Следует заметить, что все выражения внутри инструкции class выполняются при создании класса, а

не его экземпляра. Для примера создадим класс, внутри которого просто выводится сообщение:

```
class MyClass:
    """ Это строка документирования """
    print("Инструкции выполняются сразу")
```

Этот пример содержит лишь определение класса *Myclass* и не создает экземпляр класса. Как только поток выполнения достигнет инструкции *class*, сообщение, указанное в функции *print*(), будет сразу выведено.

С точки зрения пространства имен класс можно представить подобным модулю. Также как в модуле в классе могут быть свои переменные со значениями и функции. Также как в модуле у класса есть собственное пространство имен, доступ к которому возможен через имя класса:

```
class B:
    n = 5
    def adder(v):
       return v + B.n
print(B.n) # Вывод: 5
     print(B.adder(4)) # Вывод: 9
```

Однако в случае классов используется особая терминология. Имена, определенные в классе, называются **атрибутами** этого класса. В примере имена *n* и *adder* – это атрибуты класса *B*. Атрибуты-переменные называют **полями** или свойствами (в других языках понятия "поле" и "свойство" не совсем одно и то же). Полем является *n*. Атрибуты-функции называются **методами**. Методом в классе *B* является *adder*. Количество свойств и методов в классе может быть любым.

Создание атрибута класса аналогично созданию обычной переменной. Метод внутри класса создается так же, как и обычная функция, – с помощью инструкции *def*. Среди атрибутов можно выделить две группы.

Атрибуты первой группы принадлежат самому классу, а не конкретному экземпляру. Доступ к таким атрибутам вне тела класса осуществляется через точечную нотацию через объект объявления класса.

```
<Имя класса>.<Имя атрибута>
```

Если этот атрибут — метод, то по аналогии можно его вызвать.

```
<Имя класса>.<Имя метода>([<Параметры>])
```

Во второй группе находятся атрибуты, принадлежащие конкретному экземпляру. В отличии от многих других языков программирования, эти атрибуты создаются уже во время существования объекта. Чтобы создать экземпляр класса используется синтаксис:

```
<экземпляр класса> = <Название класса>([<Параметры>])
```

Когда экземпляр класса существует, можно задавать атрибуты уникальные для этого самого экземпляра:

```
<экземпляр класса>.<Имя атрибута> = <Значение>
```

И по аналогии получать доступ к уже существующим атрибутам:

```
<экземпляр класса>.<Имя атрибута>
```

При этом если не удается найти атрибут с таким именем у экземпляра, то поиск продолжается в атрибутах, принадлежащих классу. В связи с этим можно считать, что атрибуты самого класса разделяются между всеми его экземплярами.

```
class MyClass:
    x = 50          # Создаем атрибут объекта класса MyClass

c1, c2 = MyClass(), MyClass() # Создаем два экземпляра класса
c1.y = 10           # Создаем атрибут экземпляра класса c1
c2.y = 20           # Создаем атрибут экземпляра класса c2
print(c1.x, ' ', c1.y) # Вывод: 50 10
print(c2.x, ' ', c2.y) # Вывод: 50 20
```

В этом примере мы определяем класс *MyClass* и создаем атрибут объекта класса: *x*. Этот атрибут будет доступен всем создаваемым экземплярам класса. Затем создаем два экземпляра класса и добавляем одноименные атрибуты: *y*. Значения этих атрибутов будут разными в каждом экземпляре класса. Но если создать новый экземпляр (например, *c3*), то атрибут *y* в нем определен не будет. Таким образом, с помощью классов можно имитировать типы данных, поддерживаемые другими языками программирования (например, тип *struct*, доступный в языке C).

Обычно все методы класса объявляются на уровне класса, а не экземпляра. Вызов такого метода через экземпляр класса с помощью синтаксиса

```
<экземпляр класса>.<Имя метода>([<Параметры>])
```

неявно преобразуется к

```
<Имя класса>.<Имя метода>(<экземпляр класса>, [<параметры>])
```

Иными словами, методам класса в первом параметре, который необходимо указывать явно, автоматически передается ссылка на экземпляр класса. Общепринято этот параметр называть именем ***self***, хотя это и не обязательно. Доступ к атрибутам и методам класса внутри определяемого метода производится через переменную *self* с помощью точечной нотации. Обратите внимание на то, что при вызове метода не нужно передавать ссылку на экземпляр класса в качестве параметра, как это делается в определении метода внутри класса. Ссылку на экземпляр класса интерпретатор передает автоматически.

Определим класс *MyClass* с атрибутом *x* и методом *print_x()*, выводящим значение этого атрибута, а затем создадим экземпляр класса и вызовем метод:

```
class MyClass:
   def __init__(self):  # Конструктор
      self.x = 1        # Атрибут экземпляра класса
   def print_x(self):   #  self — это ссылка на экземпляр класса
      print(self.x)     # Выводим значение атрибута
c = MyClass()           # Создание экземпляра класса
# Вызываем метод print_x()
c.print_x()  # self не указывается при вызове метода
print(c.x)   # К атрибуту можно обратиться непосредственно
```

Все атрибуты класса в языке Python являются открытыми (*public*), т. е. доступными для непосредственного изменения как из самого класса, так и из других классов и из основного кода программы.

Очень важно понимать разницу между атрибутами объекта класса и атрибутами экземпляра класса. Атрибут объекта класса существует в единственном экземпляре, доступен всем экземплярам класса, его изменение можно видеть во всех экземплярах класса. Атрибут экземпляра класса хранит уникальное значение для каждого экземпляра, и изменение его в одном экземпляре класса не затронет значения одноименного атрибута в других экземплярах того же класса. Рассмотрим это на примере, создав класс с атрибутом объекта класса (x) и атрибутом экземпляра класса (y):

```
class MyClass:
   x = 10  # Атрибут объекта класса общий для всех экземпляров
   def __init__(self):
      self.y = 20   # Атрибут экземпляра класса уникальный для каждого экземпляра
c1 = MyClass()  # Создаем  первый экземпляр класса
c2 = MyClass()  # Создаем  второй экземпляр класса
#Выведем значения атрибута x, а затем изменим значение и опять произведем вывод:
print (c1.x,  c2.x)  # Вывод: 10  10
MyClass.x = 88       # Изменяем атрибут объекта (класса MyClass)
print (c1.x,  c2.x)  # Вывод: 88  88
print (c1.y,  c2.y)  # Вывод: 20 20
cl.y = 88  # Изменяем атрибут экземпляра класса c1
print (c1.y,  c2.y)  # Вывод: 88 20
```

Как видно из примера, изменение атрибута объекта класса x затронуло значение в обоих экземплярах класса. Аналогичная операция с атрибутом y изменяет значение только в экземпляре c1.

Следует также учитывать, что в одном классе могут одновременно существовать атрибут объекта и атрибут экземпляра с одним именем. Изменение атрибута объекта класса мы производили следующим образом:

```
MyClass.x = 88 # Изменяем атрибут объекта класса
```

Если после этой инструкции вставить инструкцию:

```
cl.x = 200  # Создаем атрибут экземпляра
```

то будет создан атрибут экземпляра класса, а не изменено значение атрибута объекта класса. Чтобы увидеть разницу, нужно вывести их значения:

```
print (cl.x, MyClass.x)  # 200 88
```

Метод __init__()

При создании экземпляра класса интерпретатор автоматически вызывает метод инициализации *__init__()*. Такой метод принято называть конструктором класса. Формат метода:

```
def  __init__(self[,  <Значение1>[,  . ..,  <ЗначениеN>]]):
     <Инструкции>
```

С помощью метода *__init__()* атрибутам класса можно присвоить начальные значения.

При создании экземпляра класса параметры этого метода указываются после имени класса в круглых скобках:

```
<Экземпляр класса> = <Имя класса> ([<Значение1> [,  ...,<ЗначениеN>]])
```

Пример использования метода:

```
class MyClass:
   def __init__(self,  value1,  value2):  #  Конструктор
      self.x = value1
      self.y = value2
c = MyClass(100,  300)  # Создаем экземпляр класса
print(c.x,  c.y)  #  Вывод:  100  300
```

Метод __del__()

Перед уничтожением экземпляра автоматически вызывается метод, называемый деструктором. В языке Python деструктор реализуется в виде предопределенного метода*__del__()* (листинг 13.6). Следует заметить, что метод не будет вызван, если на экземпляр класса существует хотя бы одна ссылка.

Впрочем, поскольку интерпретатор самостоятельно заботится об удалении объектов, использование деструктора в языке Python не имеет особого смысла.

12. Наследование

Наследование является, пожалуй, самым главным понятием ООП.

Предположим, у нас есть класс (например, *Class1*). При помощи наследования мы можем создать новый класс (например, *Class2*), в котором будет реализован доступ ко всем атрибутам и методам класса *Class1*:

```
class Class1:
# Базовый класс
   def func1(self):
      print("Метод  func1()  класса  Class1")
   def func2(self) :
      print("Метод func2()  класса Class1")

class Class2(Classl):
# Класс Class2 наследует класс Classl
   def func3(self):
      print("Метод func3()  класса Class2")

c = Class2()   # Создаем экземпляр класса Class2
c.func1()      # Выведет: Метод  funcl() класса  Classl
c.func2()      # Выведет: Метод  func2() класса  Classl
c.func3()      # Выведет: Метод  func3() класса  Class2
```

Как видно из примера, класс *Classl* указывается внутри круглых скобок в определении класса *Class2*. Таким образом, класс *Class2* наследует все атрибуты и методы класса *Classl*.

Класс *Classl* называется базовым или суперклассом, а класс *Class2* – производным или подклассом.

Если имя метода в классе *Class2* совпадает с именем метода класса *Classl*, то будет использоваться метод из класса *Class2*. Чтобы вызвать одноименный метод из базового класса, перед методом следует через точку написать название базового класса, а в первом параметре метода – явно указать ссылку на экземпляр класса. Рассмотрим это на примере

```
class Class1:
#  Базовый класс
   def __init__( self ):
      print("Конструктор базового класса")
   def func1(self):
      print ("Метод func1()  класса Class1")

class Class2(Class1):
#  Класс Class2 наследует класс Classl
   def __init__( self ):
      print("Конструктор производного класса")
      Class1.__init__(self)  # Вызываем конструктор базового класса
   def func1(self):
      print ("Метод func1() класса Class2")
      Class1.func1(self)  # Вызываем метод базового класса

c = Class2()  # Создаем экземпляр класса Class2
c.func1()     # Вызываем метод func1()
```

Вывод:

```
Конструктор производного класса
Конструктор базового класса
Метод func1() класса Class2
Метод func1() класса Classl
```

Обратите внимание, что конструктор базового класса автоматически не вызывается, если он переопределен в производном классе. Поэтому его нужно вызывать явно либо так, как в приведенном примере, либо используя метод *super():*

```
super().__init__()
# Вызываем конструктор базового класса
```

или так:

```
super(Class2,  self).__init__()
# Вызываем конструктор базового класса
```

При использовании функции *super()* не нужно явно передавать указатель *self* в вызываемый метод. Кроме того, в первом параметре функции *super()* указывается производный класс, а не базовый.

13. Множественное наследование

В определении класса в круглых скобках можно указать сразу несколько базовых классов через запятую. Рассмотрим пример

```
class Classl:
# Базовый класс для класса Class2
   def func1(self):
      print ("Метод func1()  класса Classl")
class Class2(Classl):
# Класс Class2 наследует класс Classl
   def func2(self):
      print("Метод func2()  класса Class2")
class Class3(Classl):
# Класс Class3 наследует класс Classl
   def func1(self):
      print("Метод func1()  класса Class3")
   def func2(self):
      print("Метод func2()  класса Class3")
   def func3(self):
      print("Метод func3()  класса Class3")
   def func4(self):
      print("Метод func4()  класса Class3")
class Class4(Class2,  Class3):
# Множественное наследование
   def func4(self):
      print( "Метод func4()  класса Class4")
c = Class4()  # Создаем экземпляр класса Class4
c.func1()  #  Вывод: Метод  func1() класса  Class3
c.func2()  #  Вывод: Метод  func2() класса  Class2
c.func3()  #  Вывод: Метод  func3() класса  Class3
c.func4()  #  Вывод: Метод  func4() класса  Class4
```

Метод *func1()* определен в двух классах: *Class1* и *Class3*. Так как вначале просматриваются все базовые классы, непосредственно указанные в определении текущего класса, то метод *func1()* будет найден в классе *Class3* (поскольку он указан в числе базовых классов в определении *Class4*), а не в классе *Classl*.

Метод *func2()* также определен в двух классах: *Class2* и *Class3*. Так как класс *Class2* стоит первым в списке базовых классов, то метод будет найден именно в нем. Чтобы наследовать метод из класса *Class3*, следует указать это явным образом:

```
class Class4(Class2,  Class3):
# Множественное наследование
   func2 = Class3.func2 # Наследуем func2()  из класса Class3,  а не из класса Class2
   def  func4(self):
      print("Метод func4()  класса Class4")
```

14. Перегрузка операторов

Перегрузка операторов позволяет экземплярам классов участвовать в обычных операциях.

Чтобы перегрузить оператор, необходимо в классе определить метод со специальным названием. В результате, для выполнения действия соответствующего данной операции будет вызываться этот метод.

Перегрузка математических операторов производится с помощью следующих методов:

Выражение	Операция	Метод
x+y	сложение	x.__add__(y)
y+x	сложение (экземпляр класса справа)	x .__radd__(y)
x+=y	сложение и присваивание	x.__iadd__(y)
x-y	вычитание	x.__sub__(y)
y-x	вычитание (экземпляр класса справа	x.__rsub__(y)
x-=y	вычитание и присваивание	x.__isub__(y)
x*y	умножение	x.__mul__(y)
y*x	умножение (экземпляр класса справа):	x.__rmul__(y)
x*=y	умножение и присваивание	x .__imul__(y)
x@y	матричное умножение	x.__matmul__(y)
y@x	матричное умножение (экземпляр класса справа)	x.__rmatmul__(y)
x@=y	Матричное умножение и присваивание	x.__imatmul__(y)
x/y	деление	x.__truediv__(y
y/x	деление (экземпляр класса справа):**)**;	x.__rtruediv__(y
x/=y	деление и присваивание	x.__itruediv__(y)
x//y	деление с округлением вниз	x.__floordiv__(y)
y/=x	деление с округлением вниз (экз. класса справа):	x.__rfloordiv __(y)
x/=y	деление с округлением вниз и присваивание	x.__ifloordiv __(y)
x%y	остаток от деления	x.__mod__(y)
y%x	остаток от деления (экземпляр класса справа):	x.__rmod__(y)
x%=y	остаток от деления и присваивание	x.__imod__(y)
x**y	возведение в степень	x.__pow__(y)
y**x	возведение в степень (экземпляр класса справа):	x.__rpow__(y)
x**=y	возведение в степень и присваивание	x.__ipow__(y)
-x	унарный минус	x.__neg__()
+x	унарный плюс	x.__pos__()
abs(x)	абсолютное значение	x.__abs__().

Пример перегрузки математических операторов:

```
class MyClass:
   def __init__(self,y):
      self.x = y
   def __add__(self,y):
      print( "Экземпляр  слева")
      return self.x + y
   def __radd__(self,y):
      print( "Экземпляр  справа")
      return self.x + y
   def __iadd__(self,y):
      print( "Сложение  с  присваиванием")
      self.x += y
      return self
c  =  MyClass(50)
print( c + 10 ) #  Вывод:  Экземпляр  слева  60
print( 20 + c ) #  Вывод:  Экземпляр  справа  70
c += 30         #  Вывод:  Сложение  с  присваиванием
print( c.x )    #  Вывод:  80
```

Перегрузка операторов сравнения производится с помощью следующих методов:

Выражение	Операция	Метод
x==y	равно	x.__eq__(y)
x!=y	не равно	x .__ne__(y)
x<y	меньше	x.__lt__(y)
x>y	больше	x.__gt__(y)
y<=x	меньше или равно	x.__le__(y)
x>=y	больше или равно	x.__ge__(y)
x in y	проверка на вхождение	x.__ contains__(y)

Пример перегрузки операторов сравнения:

```
class  MyClass:
   def  __init__(self):
      self.x = 50
      self.arr   =  [1,  2,  3,  4,  5]
   def  __eq__(self, y ):  #  Перегрузка  оператора  ==
      return self.x == y
   def __contains__(self,y) :  #  Перегрузка  оператора  in
      return y in self.arr
c  =  MyClass ()
print("Равно"  if c  ==  50  else  "Не  равно")  #  Вывод:  Равно
print("Равно"  if c  ==  51  else  "Не  равно")  #  Вывод:  Не  равно
print("Есть" if 5 in c else  "Нет")  #  Вывод:  Есть
```

Возможность перегрузки операторов обеспечивает схожесть пользовательского класса со встроенными классами Python. Ведь все встроенные типы данных Питона – это классы. В результате все объекты могут иметь одинаковые интерфейсы. Так если ваш класс предполагает обращение к элементу объекта по индексу, например a[0], то это нужно обеспечить.

Пусть будет класс-агрегат B, содержащий в списке объекты класса A:

```
class A:
   def __init__(self, arg):
      self.arg = arg
   def __str__(self):
      return str(self.arg)
class B:
   def __init__(self, *args):
      self.aList = []
      for i in args:
         self.aList.append(A(i))
group = B(5, 10, 'abc')
```

Чтобы получить элемент списка, несомненно, мы можем обратиться по индексу к полю *aList*:

```
print(group.aList[1])
```

Однако куда интереснее извлекать элемент по индексу из самого объекта, а не из его поля:

```
class B:
    def __init__(self, *args):
        self.aList = []
        for i in args:
            self.aList.append(A(i))
    def __getitem__(self, i):
        return self.aList[i]
group = B(5, 10, 'abc')
print(group.aList[1])  # вывод: 10
print(group[0])  # 5
print(group[2])  # abc
```

Это делает объекты класса B похожими на объекты встроенных в Python классов-последовательностей (списков, строк, кортежей). Здесь метод *__getitem__()* перегружает операцию извлечения элемента по индексу. Другими словами, этот метод вызывается, когда к объекту применяется операция извлечения элемента: *объект[индекс]*.

Бывает необходимо, чтобы объект вел себя как функция. Это значит, если у нас есть объект a, то мы можем обращаться к нему в нотации функции, т. е. ставить после него круглые скобки и даже передавать в них аргументы:

```
a = A()
a()
a(3, 4)
```

Метод *__call__()* автоматически вызывается, когда к объекту обращаются как к функции.

```
class Changeable:
    def __init__(self, color):
        self.color = color
    def __call__(self, newcolor):
        self.color = newcolor
    def __str__(self):
        return "%s"%self.color
canvas = Changeable("green")
frame = Changeable("blue")
canvas("red")
frame("yellow")
print(canvas, frame)
```

В этом примере с помощью конструктора класса при создании объектов устанавливается их цвет. Если требуется его поменять, то достаточно обратиться к объекту как к функции и в качестве аргумента передать новый цвет. Такой обращение автоматически вызовет метод *__call__()*, который, в данном случае, изменит атрибут *color* объекта.

15. Абстрактные методы

Абстрактные методы содержат только определение метода без реализации. Предполагается, что производный класс должен переопределить метод и реализовать его функциональность. Чтобы такое предположение сделать более очевидным, часто внутри абстрактного метода возбуждают исключение:

```
class Class1:
   def __init__(self,val):
      self.x=val
   def func(self):  # Абстрактный метод
      # Возбуждаем исключение
      raise NotImplementedError("Нельзя вызывать абстрактный метод")
class Class2(Class1):  # Наследуем абстрактный метод
   def func(self):  #  Переопределяем метод
       print(self.x)
c2 = Class2(10)
c2.func()  # Вывод:  10
c1 = Class1(20)
try:  # Перехватываем исключения
   c1.func()  # Ошибка.  Метод func()  не переопределен
except NotImplementedError as msg:
   print(msg)  # Вывод:  Нельзя вызывать абстрактный метод
```

Модуль стандартной библиотеки *abc* (аббревиатура от Abstract Base Class) предоставляет дополнительные возможности. Наследуя от класса “*abc.ABC*”, вы явно указываете, что объявляемый класс — абстрактный базовый класс, т.е. создание его экземпляров не предполагается. Такой класс лишь задаёт интерфейс, который должен быть реализован в производных классах. Если хотя бы один из специальным образом помеченных методов абстрактного базового класса не переопределен, то Python бросит ошибку *TypeError* при попытке создания экземпляра этого класса. Такое поведение позволит отловить ошибку на более раннем этапе, если программист забыл про какой-то из абстрактных методов.

```
import abc
class Class1(abc.ABC):
    def __init__(self, val):
        self.x = val
    @abc.abstractmethod # Абстрактный метод
    def func(self):
        raise NotImplementedError("Нельзя вызывать абстрактный метод")

class Class2(Class1): # Наследуем абстрактный метод
    def another_func(self): # Определяем другой метод
        print(-self.x)

class Class3(Class2):  # Наследуем два метода
    def func(self):  #  Переопределяем абстрактный метод метод
        print(self.x)

try: # Перехватываем исключения
    c = Class1(10) # Ошибка.  Метод func()  не переопределен
except TypeError as msg:
    print(msg) # вывод: Can't instantiate abstract class Class1 with abstract …

try: # Перехватываем исключения
    c = Class2(10) # Ошибка.  Метод func()  не переопределен
except TypeError as msg:
    print(msg) # вывод: Can't instantiate abstract class Class1 with abstract …

c = Class3(30)
c.func() # вывод: 30
c.another_func() # вывод: -30
```

В рассмотренном примере, экземпляры классов *Class1* и *Class2* невозможно создать, т.к. у обоих из них есть метод *func*, помеченный абстрактным с помощью декоратора *@abc.abstractmethod*. Класс *Class3* переопределяет этот метод, что позволяет создать его экземпляры.

16. Ограничение доступа к атрибутам класса

В Python нет истинно закрытых атрибутов. Какие бы усилия вы не приложили, пользователь всегда сможет получить доступ к любому атрибуту вашего класса. По этой причине в Python обычно отдают предпочтение более простому открытому коду, нежели чем усложненному, который пытается скрыть детали имплементации.

Вместо этого программисты на python стараются следовать общепринятого соглашения, что все идентификаторы, имя которых не начинается с символов нижнего подчеркивания, считаются публичными. Пользователи класса смело могут напрямую обращаться к таким идентификаторам, а разработчики берут на себя ответственность сохранять наличие и роль таких идентификаторов в ближайших обновлениях. Совокупность таких идентификаторов часто называют интерфейсом класса.

Следует обратить внимание, что идентификаторы, имена которых начинаются с двух символов нижнего подчеркивания, не видны напрямую:

```
class MyClass:
   def __init__(self, x):
          self.__x = x
          self.y = x ** 2
a = MyClass(2)
print(a.y)   # вывод 4
print(a.__x) # ошибка, объект не имеет атрибута __x
```

Тем не менее, к ним тоже можно обратиться извне добавив имя класса с предшествующим символом подчеркивания:

```
print(a._MyClass__x) # вывод: 2
```

17. Полиморфизм

В качестве примера предположим, что нужно реализовать приложение с базой данных сотрудников. Имеет смысл начать с создания универсального суперкласса, в котором определены стандартные линии поведения, общие для всех типов сотрудников в организации.

```
class Employee:
# Универсальный суперкласс сотрудников
    def computeSalary (self) : . . .  # Стандартный расчет зарплаты
```

После написания кода общего поведения можно специализировать его для каждого индивидуального типа сотрудника, отражая его отличия от нормы. То есть можно создавать подклассы, настраивающие только те фрагменты поведения, которые отличаются в зависимости от типа сотрудника; остальное поведение будет унаследовано от более универсального класса. Скажем, если с инженерами связано уникальное правило подсчета заработной платы, то можно заменить в подклассе только один метод:

```
class Engineer(Employee ) :       # Специализированный подкласс инженеров
   def computeSalary(self) : ...  # Специальный метод расчета зарплаты
```

Из-за того, что версия *computeSalary* находится ниже в дереве классов, она заместит (переопределит) универсальную версию в *Employee*. Теперь можно создавать экземпляры разновидностей классов сотрудников, к которым принадлежат реальные сотрудники, чтобы получить корректное поведение:

```
bob =  Employee()  # Стандартное поведение
sue =  Employee()  # Стандартное поведение
tom =  Engineer()  # Специальный расчет заработной платы
```

Обратите внимание, что можно создавать экземпляры любого класса в дереве (за исключением абстрактных), а не только классов в нижней части — класс, из которого вы создаете экземпляр, определяет уровень, откуда будет начинаться поиск атрибутов, и соответственно то, какие версии методов он будет задействовать. В конце концов, эти объекты могут оказаться встроенными в более крупный контейнерный объект (например,

список или экземпляр другого класса), который представляет отдел или компанию. Когда нужно будет запросить заработные платы сотрудников, они будут рассчитываться в соответствии с классами, из которых создавались объекты, благодаря принципам поиска в иерархии наследования:

```
company = [bob, sue, tom]    # Список сотрудников
for emp in company:
   print( emp.computeSalary() )  # Метод computeSalary() из соответствующего класса
```

18. Композиция

Еще одной особенностью объектно-ориентированного программирования является возможность реализовывать так называемый композиционный подход. Заключается он в том, что есть класс-контейнер, он же агрегатор, который включает в себя вызовы других классов. В результате получается, что при создании объекта класса-контейнера, также создаются объекты других классов.

Чтобы понять, зачем нужна композиция в программировании, проведем аналогию с реальным миром. Большинство биологических и технических объектов состоят из более простых частей, также являющихся объектами. Например, животное состоит из различных органов (сердце, желудок), компьютер — из различного "железа" (процессор, память).

Не следует путать композицию с наследованием, в том числе множественным. Наследование предполагает принадлежность к какой-то общности (похожесть), а композиция – формирование целого из частей. Наследуются атрибуты, т. е. возможности, другого класса, при этом, объектов непосредственно родительского класса не создается. При композиции же класс-агрегатор создает объекты других классов.

Рассмотрим на примере реализацию композиции в Python. Пусть, требуется написать программу, которая вычисляет площадь обоев для оклеивания помещения. При этом окна, двери, пол и потолок оклеивать не надо.

Прежде, чем писать программу, займемся объектно-ориентированным проектированием. То есть разберемся, что к чему. Комната – это прямоугольный параллелепипед, состоящий из шести прямоугольников. Его площадь представляет собой сумму площадей составляющих его прямоугольников. Площадь прямоугольника равна произведению его длины на ширину.

По условию задачи обои клеятся только на стены, следовательно, площади верхнего и нижнего прямоугольников нам не нужны. Кроме того, надо будет вычесть общую площадь дверей и окон, поскольку они не оклеиваются.

Можно выделить три типа объектов – окна, двери и комнаты. Получается три класса. Окна и двери являются частями комнаты, поэтому пусть они входят в состав объекта-помещения.

Для данной задачи существенное значение имеют только два свойства – длина и ширина. Поэтому классы «окна» и «двери» можно объединить в один. Если бы были важны другие свойства (например, толщина стекла, материал двери), то следовало бы для окон создать один класс, а для дверей – другой. Пока обойдемся одним, и все что нам нужно от него – площадь объекта:

```
class WinDoor:
   def __init__(self, x, y):
      self.square = x * y
```

Класс "комната" – это класс-контейнер для окон и дверей. Он должен содержать экземпляры класса *WinDoor*.

Хотя помещение не может быть совсем без окон и дверей, но может быть чуланом, дверь которого также оклеивается обоями. Поэтому имеет смысл в конструктор класса вынести только размеры самого помещения, без учета элементов "дизайна", а последние

добавлять вызовом специально предназначенного для этого метода, который будет добавлять объекты-компоненты в список.

```
class Room:
   def __init__(self, x, y, z):
      self.square = 2 * z * (x + y)
      self.wd = []
   def add_wd(self, w, h):
      self.wd.append(WinDoor(w, h))
   def work_surface(self):
      new_square = self.square
      for i in self.wd:
         new_square -= i.square
      return new_square
#-----------------------------------
r1 = Room(6, 3, 2.7)
print(r1.square)          # вывод: 48.6
r1.add_wd(1, 1)
r1.add_wd(1, 1)
r1.add_wd(1, 2)
print(r1.work_surface())  #вывод: 44.6
```

19. Статические методы

Ранее было сказано, с определенным допущением классы можно рассматривать как модули, содержащие переменные со значениями и функции. Только здесь переменные называются полями или свойствами, а функции – методами. Вместе поля и методы называются атрибутами. Когда метод применяется к объекту, этот экземпляр передается в метод в качестве первого аргумента:

```
class A:
   def meth(self):
      print('meth')

a = A()
a.meth()  # для объкта a (экземпляра класса A) вызываем метод meth(a)
A.meth(a) # вызываем метод принадлежащий классу A и передаем ему экземпляр a
```

Т.е. *a.meth()* на самом деле преобразуется к *A.meth(a)*, то есть мы идем к "модулю A" и в его пространстве имен ищем атрибут *meth*. Там оказывается, что meth это функция, принимающая один обязательный аргумент. Тогда ничего не мешает сделать так:

```
class A:
   def meth(self):
      print('meth')
A.meth(10)
```

В таком "модульном формате" вызова методов передавать объект-экземпляр именно класса A совсем не обязательно. Что делать, если возникает необходимость в методе, который не принимал бы объект данного класса в качестве аргумента? Да, мы можем объявить метод вообще без параметров и вызывать его только через класс:

```
class A:
   def meth():
      print('meth')
a = A()
A.meth()  # вызываем метод без параметров принадлежащий классу A
a.meth()  # Ошибка, в метод без параметров передается аргумент self
```

Получается странная ситуация. Ведь meth() вызывается не только через класса, но и через порожденные от него объекты. Однако в последнем случае всегда будет возникать ошибка. Кроме того, может понадобиться метод с параметрами, но которому не надо передавать экземпляр данного класса.

Для таких ситуаций предназначены статические методы. Эти методы могут вызываться через объекты данного класса, но сам объект в качестве аргумента в них не передается. В Python острой необходимости в статических методах нет. Если нам нужна просто какая-нибудь функция, мы можем определить ее вне класса. Единственное достоинство в том, что функция оказывается в пространстве имен этого класса.

Статические методы в Python реализуются с помощью специального декоратора *@staticmethod*:

```
class A:
   @staticmethod
   def meth():
       print('meth')
A.meth()   # вызываем статический метод (без параметров) принадлежащий классу A
a = A()
a.meth()   # вызываем статический метод без параметров через экземпляр класса A
```

Вообще, если в теле метода не используется self, то есть ссылка на конкретный объект, следует задуматься, чтобы сделать метод статическим.

Пусть у нас будет класс "Цилиндр". При создании объектов от этого класса у них заводятся поля высота и диаметр, а также площадь поверхности. Вычисление площади можно поместить в отдельную статическую функцию. Она вроде и относится к цилиндрам, но, с другой стороны, само вычисление объекта не требует и может быть использовано где угодно.

```
from math import pi
class Cylinder:
    @staticmethod
    def make_area(d, h):
        circle = pi * d**2 / 4
        side = pi * d * h
        return circle*2 + side
    def __init__(self, diameter, high):
        self.dia = diameter
        self.h = high
        self.area = self.make_area(diameter, high)
a = Cylinder(1, 2)
print(a.area)
print(a.make_area(2, 2))
```

В примере вызов *make_area()* за пределами класса возможен в том числе через экземпляр. При этом понятно, в данном случае свойство *area* самого объекта *a* не меняется. Мы просто вызываем функцию, находящуюся в пространстве имен класса.

20. Примеры объектно-ориентированных программ на Python

В ООП очень важно предварительное проектирование. В общей сложности можно выделить следующие этапы разработки объектно-ориентированной программы:

1. Формулирование задачи.
2. Определение объектов, участвующих в ее решении.
3. Проектирование классов, на основе которых будут создаваться объекты. В случае необходимости установление между классами наследственных связей.
4. Определение ключевых для данной задачи свойств и методов объектов.
5. Создание классов, определение их полей и методов.
6. Создание объектов.
7. Решение задачи путем организации взаимодействия объектов.

Далее приведены примеры классов в порядке возрастания сложности. Сначала простые классы. Далее – классы демонстрирующие наследование, полиморфизм и композицию.

- ### *Класс рациональных дробей*

Простой класс, представляющий рациональную дробь (*num* – числитель, *den* – знаменатель). Класс содержит конструктор и перегруженные методы умножения и деления (дроби на дробь и дроби на целое число). Метод создания случайной дроби из заданного диапазона целых чисел объявлен как статический.

Следует отметить, что в языке имеется готовый тип *Fraction* в модуле *fractions*. И данный пример нужно рассматривать только как образец для создания собственных классов.

```
from math import gcd
from random import randint

class My_Fraction:
    def __init__(self, num, den):
        if num != 0 and den != 0:
            k = gcd(num, den) # находим НОД
            self.num = num // k  # числитель
            self.den = den // k  # знасенатель
        else:
            raise ValueError

    @staticmethod
    def generate(num_min, num_max, den_min, den_max):
        return My_Fraction(randint(num_min, num_max), randint(den_min, den_max))

    def __str__(self):           # Метод преобразования дроби в строку
        return f'{self.num}/{self.den}'

    def __mul__(self, other):    # Умножение дробей
        if isinstance(other,My_Fraction): # перегрузка умножения на дробь
            return My_Fraction(self.num * other.num, self.den * other.den)
        if isinstance(other,int):  # перегрузка умножения на целое число
            return My_Fraction(self.num * other, self.den)
        return self                # для остальных типов возвращаем значение самого
объекта

    def __truediv__(self, other): # Деление дробей
        if isinstance(other,My_Fraction):  # перегрузка деления на дробь
            return My_Fraction(self.num * other.den, self.den * other.num)
        if isinstance(other,int):  # перегрузка деления на целое число
            return My_Fraction(self.num, self.den*other)
        raise TypeError            # для остальных типов вызываем исключение
#---------------------------------------------------------------------
# Список из 5 случайных дробей:
a = [My_Fraction.generate(1, 9, 1, 9) for i in range(5)]
for f in a:
    b = My_Fraction.generate(1, 9, 1, 9)  # дробь для правого операнда
    cm = f * b
    print(f'{f} * {b} = {cm}') # пример умножения на дробь
    cd = f / b
    print(f'{f} / {b} = {cd}') # пример деления на дробь
    n=randint(1, 9)  # число для правого операнда
    cm = f * n
    print(f'{f} * {n} = {cm}') # пример умножения на число
    cd = f / n
    print(f'{f} / {n} = {cd}') # пример деления на число
```

- *Класс «Студент»*

Класс содержит имя студента *full_name*, номер группы *group_number* и список полученных оценок p*rogress*. В программе вводится список студентов. Далее список сортируется по имени, потом выводятся студенты, имеющие неудовлетворительные оценки.

```
class Student:
    def __init__(self,full_name="", group_number=0, progress=[]): # конструктор
        self.full_name = full_name  # имя
        self.group_number = group_number # номер группы
        self.progress = progress   # оценки
    def __str__(self):   # печатаемое представление экземпляра класса
        txt = 'Студент: ' + self.full_name + ' Группа: ' + self.group_number
        txt += ' Оценки:'
        for x in self.progress:
           txt += ' ' + str(x) # добавляем список оценок
        return txt
#--------------------------------------------------------------------------------
def SortParam(st): # функция определяющая атрибут для сортировки
   return st.full_name
#--------------------------------------------------------------------------------
st_size = 5  # количество студенов

students = [] # создание пустого списка
for i in range(st_size): # цикл для ввода  st_size студентов
    print("Введите полное имя студента: ")
    full_name = input() # ввод фамилии
    print("Введите номер группы: ")
    group_number = input() # ввод группы
    n=5
    print('Введите ',n,' оценок в столбик: ') # у каждого студента n оценок
    progress = []
    for i in range(n):
        score = int(input()) # ввод оценок
        progress.append(score) # добавление оценок
    # создание экзепляра класса Student:
    st = Student(full_name, group_number, progress)
    students.append(st) # добавление экземпляра в список

print("Students list:")
for st in students: # вывод полного списка студентов
    print(st)

# сортировка по фамилии, ключ сортировки определяется функцией SortParam:
students = sorted(students, key=SortParam)

print("Sorted students:")
for st in students: #  вывод отсортированного списка
    print(st)

print("bad students:")
n=0 # счетчик количества неуспевающих
for st in students: #  вывод неуспевающих
   for val in st.progress:
      if val<3 : # есть плохая оценка
         print(st) # выводим студента с плохой оцекой
         n += 1
         break
if n == 0:
    print("no matches were found.")
```

- ### *Виртуальная модель процесса обучения*

Пусть необходимо разработать виртуальную модель процесса обучения. В программе должны быть объекты-ученики, учитель, кладезь знаний.

Потребуется три класса – "учитель", "ученик", "данные". Учитель и ученик во многом похожи, оба – люди. Значит, их классы могут принадлежать одному надклассу "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): #конструктор
        self.info = list(info)
    def __getitem__(self, i): # перегрузка [] для извлечения  элемента из Data
        return self.info[i]
class Teacher:
    def __init__(self): #конструктор
        self.work = 0  # количество учеников
    def teach(self, info, pupil): # обучение данными из info учеников pupil
        for i in pupil:
            i.take(info) # учим ученика i
            self.work += 1 # количество учеников увеличилось на 1
class Pupil:
    def __init__(self):  #конструктор
        self.knowledge = [] # список полученных знаний
    def take(self, info): # получение знания
        self.knowledge.append(info)
```

В класс *Teacher* добавлено свойство экземпляров *work*, чтобы подсчитывать количество проделанной учителем работы.

Теперь посмотрим, как объекты этих классов могут взаимодействовать между собой:

```
lesson = Data(['class', 'object', 'inheritance', 'polymorphism', 'encapsulation'])
marIvanna = Teacher()
vasy = Pupil()
pety = Pupil()
marIvanna.teach(lesson[2], [vasy, pety]) # учить обоих знанию lesson[2]
marIvanna.teach(lesson[0], [pety])   # учить pety знанию lesson[0]
print(vasy.knowledge) # вывод: ['inheritance']
print(pety.knowledge) # вывод: ['inheritance', 'class']
```

- ### *Игра-стратегиия «Солдаты и герои»*

В некой игре-стратегии есть солдаты и герои. У всех есть свойство, содержащее уникальный номер объекта, и свойство, в котором хранится принадлежность команде. У солдат есть метод "иду за героем", который в качестве аргумента принимает объект типа "герой". У героев есть метод увеличения собственного уровня.

В основной ветке программы создается по одному герою для каждой команды. В цикле генерируются объекты-солдаты. Их принадлежность команде определяется случайно.

Солдаты разных команд добавляются в разные списки. Измеряется длина списков солдат противоборствующих команд и выводится на экран. У героя, принадлежащего команде с более длинным списком, увеличивается уровень.

Отправляем одного из солдат следовать за первым героем и выводим их идентификационные номера.

```
from random import randint
class Person:
    count = 0
    def __init__(self, c):
        self.id = Person.count
        Person.count += 1
        self.command = c
class Hero(Person):
    def __init__(self, c):
        Person.__init__(self, c)
        self.level = 1
    def up_level(self):
        self.level += 1
class Soldier(Person):
    def __init__(self, c):
        Person.__init__(self, c)
        self.my_hero = None
    def follow(self, hero):
        self.my_hero = hero.id
h1 = Hero(1)  # первый герой
h2 = Hero(2)  # второй герой
army1 = []    # первая армия
army2 = []    # вторая армия
for i in range(20):
    n = randint(1, 2)
    if n == 1:
        army1.append(Soldier(n)) # добавление солдата в первую армию
    else:
        army2.append(Soldier(n)) # добавление солдата во вторую армию
print(len(army1), len(army2))    # численность армий
if len(army1) > len(army2):  # повышение уровня героя для команды с большей
численностью
    h1.up_level()
elif len(army1) < len(army2):
    h2.up_level()
army1[0].follow(h1) # первому солдату следовать за 1 героем
print(army1[0].id, h1.id) ) # номера первого солдата и первого героя
```

- *Класс «Битва»*

В классе *Battle* реализована композиция: он включает два объекта типа *Soldier*.

```
from random import randint
class Soldier: # класс описывающий одного солдата
    def __init__(self,name='Noname',health = 100): # конструктор
       self.name = name     # задаем имя воина
       self.health = health # задаем начальное здоровье
    def set_name(self, name):
       self.name = name     # есть возможность поменять имя
    def make_kick(self, enemy): # метод моделирующий атаку на солдата enemy
       enemy.health -= 20  # при атаке здоровье врага уменьшаем на 20
       if enemy.health<0 :
          enemy.health = 0
       self.health += 10   # а собственное здоровье увеличиваем на 10
       print(self.name, "бьет", enemy.name) # выводим кто кого бьет
       print('%s = %d' % (enemy.name, enemy.health)) # выводим состояние врага
#--------------------------------------------------------------------------
class Battle:
    def __init__(self,u1,u2):   # конструктор
    # композиция: класс включает двух солдат u1 и u2
       self.u1 = u1
       self.u2 = u2
       self.result = "Сражения не было" # строка для хранения состояния сражения
    def battle(self): # метод моделирующий сражение
       while self.u1.health > 0 and self.u2.health > 0:
          n = randint(1, 2) # определяем, кто атакует
          if n == 1:
             self.u1.make_kick(self.u2) # если атакует первый
          else:
             self.u2.make_kick(self.u1) # если атакует второй
       if self.u1.health > self.u2.health:# определяем, кто победил
          self.result = self.u1.name + " ПОБЕДИЛ"
       elif self.u2.health > self.u1.health:
          self.result = self.u2.name + " ПОБЕДИЛ"
    def who_win(self): # вывод результата
        print(self.result)
#--------------------------------------------------------------------------
first = Soldier('Mr. First',140) # созаем 1 солдата с именем Mr. First и здоровьем
140
second = Soldier()  # созаем 2 солдата с паметрами по умолчанию
second.set_name('Mr. Second') # меняем имя 2 солдата
b = Battle(first,second) # создаем объект Battle
b.battle()  # запускаем сражение
b.who_win()  # выводим итог
```

- *Класс «Колода карт»*

Класс *DeckOfCards* содержит список карт. Конструктор класса Card инициализирует значения масти и номера из списков *NumsList* и *MastList*, которые объявлены как общие атрибуты класса.

```
import time
import random;
class Card(): # класс Карта
   NumsList = ["Джокер",'2','3','4','5','6','7','8','9','10',
               "Валет","Дама","Король","Туз"]
   MastList = ["пик","крестей","бубей","червей"]
   def __init__(self, i, j):# конструктор
      self.Mastb = self.MastList[i-1]; # карта
      self.Num   = self.NumsList[j-1]; # масть
#---------------------------------------
class DeckOfCards(): # класс Колода карт
   def __init__(self): # конструктор
      self.deck = [None] * 56; # список из 56 карт
      k = 0;
      for i in range(1, 4 + 1):
         for j in range(1, 14 + 1):
            self.deck[k] = Card(i, j); # очередная карта
            k += 1;
   def shuffle(self): # перемешивание карт
      random.shuffle(self.deck);
   def get(self, i): # вытаскивание i-й карты из колоды
      if 0 <= i <=55 :
         answer = self.deck[i].Num;
         answer += " ";
         answer += self.deck[i].Mastb;
      else :
         answer = "В колоде только 56 карт"
      return answer;
#---------------------------------------
deck = DeckOfCards(); # создали колоду
deck.shuffle();       # перемешали
print('Выберите карту из колоды в 56 карт:');
n=int(input())
if n<=56 :
   card = deck.get(n-1);
   print('Вы взяли карту: ', card, end='.\n');
else :
   print("В колоде только 56 карт")
```

- *Класс «Паспорт»*

Класс *ForeignPassport* является производным от класса *Passport*. Метод *PrintInfo* существует в обоих классах. *PassportList* представляет собой список, содержащий объекты обоих классов. Вызов метода *PrintInfo* для каждого элемента списка демонстрирует его полиморфное поведение.

```
class Passport():
    def __init__(self, first_name, last_name, country, date_of_birth,
numb_of_pasport):
        self.first_name = first_name
        self.last_name = last_name
        self.date_of_birth = date_of_birth
        self.country = country
        self.numb_of_pasport = numb_of_pasport

    def PrintInfo(self):
       print("\nFullname: ",self.first_name, " ",self.last_name)
       print("Date of birth: ",self.date_of_birth)
       print("County: ",self.country)
       print("Passport number: ",self.numb_of_pasport)

class ForeignPassport(Passport):
    def __init__(self, first_name, last_name, country, date_of_birth,
numb_of_pasport,visa):
        super().__init__(first_name, last_name, country, date_of_birth,
numb_of_pasport)
        self.visa = visa

    def PrintInfo(self):
        super().PrintInfo()
        print("Visa: ",self.visa)

PassportList=[]
request = ForeignPassport('Ivan', 'Ivanov', 'Russia', '12.03.1967', '123456789',
'USA')
PassportList.append(request)

request = Passport('Иван', 'Иванов', 'Россия', '12.03.1967', '45001432')
PassportList.append(request)

request = ForeignPassport('Peter', 'Smit', 'USA', '01.03.1990', '21435688',
'Germany')
PassportList.append(request)

for emp in PassportList:
   emp.PrintInfo()
```

- *Класс «Склад оргтехники»*

Классы *Printer, Scaner* и *Xerox* являются производными от класса *Equipment.* Метод *str()* перегружен только в классе *Printer,* для остальных используется метод из базового класса. Метод *action()* перегружен для всех производных классов. Вызов этих методов для каждого элемента списка демонстрирует их полиморфное поведение.

```
class Equipment:
    def __init__(self, name, make, year):
        self.name = name # производитель
        self.make = make # модель
        self.year = year # год выпуска
    def action(self):
        return 'Не определено'
    def __str__(self):
        return f'{self.name} {self.make} {self.year}'
#-----------------------------------------------
class Printer(Equipment):
    def __init__(self, series, name, make, year):
        super().__init__(name, make, year)
        self.series = series # серия
    def __str__(self):
        return f'{self.name} {self.series} {self.make} {self.year}'
    def action(self):
        return 'Печатает'
#-----------------------------------------------
class Scaner(Equipment):
    def __init__(self, name, make, year):
        super().__init__(name, make, year)
    def action(self):
        return 'Сканирует'
#-----------------------------------------------
class Xerox(Equipment):
    def __init__(self, name, make, year):
        super().__init__(name, make, year)
    def action(self):
        return 'Копирует'
#-----------------------------------------------
sklad = []
# создаем объект сканер и добавляем
scaner = Scaner('Mustek','BearPow 1200CU', 2010)
sklad.append(scaner)
# создаем объект ксерокс и добавляем
xerox = Xerox('Xerox','Phaser 3120', 2019)
sklad.append(xerox)
# создаем объект принтер и добавляем
printer = Printer("1200",'hp', 'Laser Jet', 2018)
sklad.append(printer)
# выводим склад
print("На складе имеются:")
for x in sklad:
   print(x,end=' ')
   print(x.action())
# забираем со склада все принтеры
for x in sklad:
   if isinstance(x,Printer):
      sklad.remove(x)
# выводим склад
print("\nНа складе осталось:")
for x in sklad:
   print(x,end=' ')
   print(x.action())
```

- *Задача трёх тел*

Задача трех тел — одна из задач небесной механики, состоящая в определении относительного движения трёх тел (материальных точек), взаимодействующих по закону тяготения Ньютона. В любой момент времени на каждое из тел действуют силы гравитационного притяжения остальных двух тел. Сила, с которой первое из тел притягивает второе, выражется формулой

$$\overrightarrow{F_{12}} = -G\frac{m_1 m_2 \overrightarrow{r_{12}}}{r_{12}^3},$$

где G — гравитационная постоянаая, m_1 и m_2 — массы тел, $\overrightarrow{r_{12}} = \overrightarrow{r_2} - \overrightarrow{r_1}$ — вектор от первого к тела к второму, где положение тел в пространстве задается радиус векторами $\overrightarrow{r_1}$ и $\overrightarrow{r_2}$. Остальные силы попарного взаимодействия можно получить заменной индексов. Итого, движение всех трел задаётся системой динамических уравнений

$$\begin{cases} m_1\overrightarrow{a_1} = \overrightarrow{F_{21}} + \overrightarrow{F_{31}} \\ m_2\overrightarrow{a_2} = \overrightarrow{F_{12}} + \overrightarrow{F_{32}} \\ m_3\overrightarrow{a_3} = \overrightarrow{F_{13}} + \overrightarrow{F_{23}} \end{cases}.$$

Такая система уравнения не имеет аналитического решения в общем случае, но можно решить её приближенно, введя малый шаг по времени Δt и используя следующую схему обновления положения и скорости тела в следующий момент времени

$$\overrightarrow{v(t+\Delta t)} = \overrightarrow{v(t)} + \overrightarrow{a(t)}\Delta t,$$
$$\overrightarrow{r(t+\Delta t)} = \overrightarrow{r(t)} + \overrightarrow{v(t)}\Delta t.$$

Рассмотрим решение задачи трех тел в объектно ориентированном стиле. Первое, что бросается в глаза — многие величины описываются векторами в трехмерном пространстве. Полезно в таком случае реализовать класс трехмерных векторов и определить для него операции сложения, вычитания, вычисления его нормы, домножения на скаляр и другие. В приведенных ниже примерах предполагается, что в программе определен класс `Vector3D` удовлетворяющий всем перечисленным свойствам, а реализация такого класса оставляется читею в качестве упражнения (смотри задачи для самостоятельного решения).

Класс *Body* описывает небесное тело. В рамках нашей задачи тело задаётся его массой, положением пространства (радиус вектор) и вектором скорости. Метод *update* рассчитывает изменение положения тела в пространстве и его вектоа скорости под действием заданной силы за время Δt по описанной выше схеме.

```
class Body:
    def __init__(self, mass, position, velocity): # конструктор
        self.mass = mass # масса
        self.position = position # радиус вектор положения в пространстве
        self.velocity = velocity # вектор скорости

    def update(self, force, dt): #
        "Обновление положения и скорости под действием силы force"
        self.velocity += force * dt / self.mass
        self.position += self.velocity * dt
```

Функция *gravity_force* принимает на вход два небесных тела (экземпляра класса *Body)* и возвращает вектор силы, с которой первое тело притягивает второе.

```
def gravity_force(A, B):
    r_AB = B.position - A.position
    return - G*A.mass*B.mass*r_AB / (r_AB.norm() ** 3)
```

Класс *ThreeBodyProblem* инкапслурует в себе три небесных тела, которых принимает в конструкторе. Метод *do_step* вычисляет силы, действующие на каждое из тел, методом *compute_forces* и обновляет положения и скорости этих тел. Метод *compute_trajectories* вычисляет и возвращает траектории движения всех трех тел на интервале времени $t\epsilon[0, T_{final}]$.

```
class ThreeBodyProblem:
    def __init__(self, body1, body2, body3):
        self.body1 = body1
        self.body2 = body2
        self.body3 = body3

    def compute_forces(self):
        # вычисление сил попарного взаимодействия
        F_12 = gravity_force(self.body1, self.body2)
        F_13 = gravity_force(self.body1, self.body3)
        F_23 = gravity_force(self.body2, self.body3)
        # противоположные силы
        F_21, F_31, F_32 = -F_12, -F_13, -F_23
        # возвращаем кортеж результирующих сил
        return F_21 + F_31, F_32 + F_12, F_13 + F_23

    def do_step(self, dt): # сделать шаг по времени
        F_1, F_2, F_3 = self.compute_forces() # рассчет силы
        self.body1.update(F_1, dt)
        self.body2.update(F_2, dt)
        self.body3.update(F_3, dt)

    def get_positions(self):
        return [self.body1.position, self.body2.position,
self.body3.position]

    def compute_trajectories(self, dt, t_final):
        t = 0 # инициализации времени
        trajectories = [self.get_positions()] # начальные положений
        while t < t_final:
            self.do_step(dt) # шаг по времени
            trajectories.append(self.get_positions()) # обновленные положения
            t += dt # приращение времени
        return trajectories
```

21. Задания для самостоятельного решения

(1) *Класс* Vector3D

Экземляр класса задается тройкой координат в трехмерном пространстве (x,y,z).
Обязательно должны быть реализованы методы:
– приведение вектора к строке с выводом кооржинат (метод __str __),
– сложение векторов оператором `+` (метод __add__),
– вычитание векторов оператором `-` (метод __sub__),
– скалярное произведение оператором `*` (метод __mul__),
– умножение и деление на скаляр операторами `*` и `/` (метод __mul__ и __truediv__),
– векторное произведение оператором `@` (метод __matmul__),
– вычисление длинны вектора методом `norm`.

Пример

```
v1 = Vector3D(4, 1, 2)
print(v1)
v2 = Vector3D()
v2.read()
v3 = Vector3D(1, 2, 3)
v4 = v1 + v2
print(v4)
a = v4 * v3
print(a)
v4 = v1 * 10
print(v4)
v4 = v1 @ v3
print(v4)
```

(2) Класс «Прямоугольный треугольник»

Класс содержит два действительных числа – стороны треугольника. и включает следующие методы:
– увеличение/уменьшение размера стороны на заданное количество процентов;
– вычисление радиуса описанной окружности,
– вычисление периметра,
– определение значений углов.

(3) Класс «Одномерный масив» TArray

Класс содержит поле для задания количества элементов и поле для хранения элементов массива.
Методы:
– конструктор без параметров, конструктор с параметрами ,конструктор копирования,
– ввод и вывод даных,
– поиск максимального и минимального элементов,
– сортировка массива,
– поиск суммы элементов
– перегрузка оператора + (добавление элемента)
– перегрузка оператора * умножение элементов массива на число

(4) Класс «Автобус».

Класс содержит свойства:
– speed (скорость),
–capacity (максимальное количество пассажиров),
– maxSpeed (максимальная скорость),
– passengers (список имен пассажиров),
– hasEmptySeats (наличие свободных мест),

– seats (словарь мест в автобусе);
методы:
– посадка и высадка одного или нескольких пассажиров,
– увеличение и уменьшение скорости на заданное значение.
– операции "in", "+=" и "−=" (посадка и высадка пассажира(ов) с заданной фамилией)

(5) Класс «Снежинки» Snow

Класс содержит целое число - количество снежинок.
Класс включает методы перегрузки арифметических операторов сложения, вычитания, умножения и деления. Код этих методов должен выполнять увеличение или уменьшение количества снежинок на число n или в n раз.

Класс также включает метод makeSnow(), который принимает сам объект и число снежинок в ряду, а возвращает строку вида
"*****\n*****\n*****…",
где количество снежинок между '\n' равно переданному аргументу, а количество рядов вычисляется, исходя из общего количества снежинок.

(6) Класс «Снежинка» (SnowFlake)

При инициализации класс принимает целое нечетное число – сторону квадрата, в который вписана снежинка.
Методы:
– thaw() – таять, при этом на каждом шаге пропадают крайние звездочки со всех сторон; параметр показывает, сколько шагов прошло.
– freeze(n) – замораживаться, при этом сторона квадрата, в который вписана снежинка, увеличивается на 2 * n, одновременно добавляются звездочки в нужных местах, чтобы правило соблюдалось.
– thicken() – утолщаться, ко всем линиям звездочек с двух сторон добавляются параллельные (если перед этим снежинка таяла, то теперь звездочки восстанавливаются).
– show() – показывать (рисуется снежинка в виде квадратной матрицы со звездочками и дефисами в пустых местах).

(7) Класс «Robot»

Класс инициализируется начальными координатами – положением Робота на плоскости, обе координаты заключены в пределах от 0 до 100.

Робот может передвигаться на одну клетку вверх (N), вниз (S), вправо (E), влево (W). Выйти за границы плоскости Робот не может.

Метод move() принимает строку – последовательность команд перемещения робота, каждая буква строки соответствует перемещению на единичный интервал в направлении, указанном буквой. Метод возвращает список координат – конечное положение Робота после перемещения.

Метод path() вызывается без аргументов и возвращает список координат точек, по которым перемещался Робот при последнем вызове метода move. Если метод не вызывался, возвращает список с начальным положением Робота.

(8) Класс «Темы» (Themes)

Экземпляру класса при инициализации передается аргумент – список тем для разговора.
Класс реализует методы:
– add_theme(value) – добавить тему в конец;
– shift_one() – сдвинуть темы на одну вправо (последняя становится первой, остальные сдвигаются);
– reverse_order() – поменять порядок тем на обратный;

– get_themes() – возвращает список тем;
– get_first() – возвращает первую тему.
Пример 1
Ввод:
```
tl = Themes(['weather', 'rain'])
tl.add_theme('warm')
print(tl.get_themes())
tl.shift_one()
print(tl.get_first())
```
Вывод:
('weather', 'rain', 'warm')
warm
Пример 2
Ввод:
```
tl = Themes(['sun', 'feeding'])
tl.add_theme('cool')
tl.shift_one()
print(tl.get_first())
tl.reverse_order()
print(tl.get_themes())
```
Вывод:
cool
('feeding', 'sun', 'cool')

(9) Класс «ПчёлоСлон» (BeeElephant)

Экземпляр класса инициализируется двумя целыми числами: первое относится к пчеле, второе – к слону.
Класс реализует следующие методы:
– fly() – может летать – возвращает True, если часть пчелы не меньше части слона, иначе False;
– trumpet() – трубить – если часть слона не меньше части пчелы, возвращает строку: “tu-tu-doo-doo!”, иначе “wzzzzz”.
– eat(meal, value) – есть – может есть только нектар (nectar) или траву (grass). Если съедает нектар, то из части слона вычитается количество съеденного, пчеле добавляется, иначе наоборот: у пчелы вычитается, а слону добавляется. Не может увеличиться выше 100 и уменьшиться меньше 0;
– get_parts() – возвращает список из значений: [часть пчелы, часть слона].
Пример 1
Ввод:
```
be = BeeElephant(3, 2)
print(be.fly())
print(be.trumpet())
be.eat('grass', 4)
print(be.get_parts())
```
Вывод:
True
wzzzzz
(0, 6)
Пример 2
Ввод:
```
be = BeeElephant(13, 87)
print(be.fly())
print(be.trumpet())
be.eat('nectar', 90)
print(be.trumpet())
print(be.get_parts())
```
Вывод:

False
tu-tu-doo-doo!
wzzzzz
(100, 0)

---

(10) Класс «Разговор» (Talking)

Экземпляр класса инициализируется с аргументом name – именем котенка. Класс реализует методы:
– to_answer() – ответить: котенок через один раз отвечает да или нет, начинает с да. Метод возвращает “moore-moore”, если да, “meow-meow”, если нет. Одновременно увеличивается количество соответствующих ответов;
– number_yes() – количество ответов да;
– number_no() – количество ответов нет.
Пример 1
Ввод:

```
tk = Talking('Pussy')
print(tk.to_answer())
print(tk.to_answer())
print(tk.to_answer())
print(f'{tk.name} says "yes" {tk.number_yes()} times, "no" {tk.number_no()}
times')
```

Вывод:
moore-moore
meow-meow
moore-moore
Pussy says "yes" 2 times, "no" 1 times
Пример 2
Ввод:

```
tk = Talking('Pussy')
tk1 = Talking('Barsik')
print(tk.to_answer())
print(tk1.to_answer())
print(tk1.to_answer())
print(tk1.to_answer())
print(f'{tk.name} says "yes" {tk.number_yes()} times, "no" {tk.number_no()}
times')
print(f'{tk1.name} says "yes" {tk1.number_yes()} times, "no"
{tk1.number_no()} times')
```

Вывод:
moore-moore
moore-moore
meow-meow
moore-moore
Pussy says "yes" 1 times, "no" 0 times
Barsik says "yes" 2 times, "no" 1 times

---

(11) Класс «Воздушный Замок» (AirCastle)

Экземпляр класса инициализируется с аргументами:
– высота;
– количество составляющих облаков;
– цвет.
Класс должен реализовывать методы:
– change_height(value) – изменить высоту на value, может уменьшаться только до нуля;
– сложить с числом, добавляется n облаков к замку, одновременно увеличивается высоту на n // 5;
– экземпляр класса можно вызвать с аргументом – целым числом, означающим

прозрачность облаков; метод возвращает значение видимости замка, рассчитанное по формуле: высота // прозрачность * количество облаков;
__str__ – возвращает строковое представление в виде:
“The AirCastle at an altitude of <высота> meters is <цвет> with <количество облаков> clouds”.
– экземпляры можно сравнивать: сначала по количеству облаков, затем по высоте, затем по цвету по алфавиту; для этого нужно реализовать методы сравнения: >, <, >=, <=, ==, !=.

---

(12) Класс Добрый Ифрит (GoodIfrit)

Экземпляр класса инициализируется с аргументами: высота, имя, доброта.
Класс должен реализовывать функциональность
– change_goodness(value) – менять доброту на указанную величину; не может стать отрицательной, в этом случае становится равной 0;
– к экземпляру класса можно прибавить число: (gi1 = gi + number), создается новый экземпляр с высотой, большей на величину number, остальные характеристики те же;
– экземпляр класса можно вызвать с аргументом, возвращается значение:
аргумент * доброта // высота
__str__() – возвращает строку вида:
“Good Ifrit <имя>, height <высота>, goodness <доброта>”
– экземпляры можно сравнивать между собой: сначала по доброте, затем по высоте, затем по имени по алфавиту; для этого нужно реализовать методы сравнения:
<, >, <=, >=, ==, !=.
Ввод:

```
gi = GoodIfrit(80, "Hazrul", 3)
gi.change_goodness(4)
print(gi)
gi1 = gi + 15
print(gi1)
print(gi(31))
```

Вывод:
Good Ifrit Hazrul, height 80, goodness 7
Good Ifrit Hazrul, height 95, goodness 7
2
Ввод:

```
gi = GoodIfrit(80, "Hazrul", 3)
gi1 = GoodIfrit(80, "Dalziel", 1)
print(gi < gi1)
gi1.change_goodness(2)
print(gi > gi1)
print(gi, gi1, sep='\n')
```

Вывод:
False
True
Good Ifrit Hazrul, height 80, goodness 3
Good Ifrit Dalziel, height 80, goodness 3

---

(13) Класс «Волшебник» (Wizard)

Экземпляр класса при инициализации принимает аргументы:
– имя;
– рейтинг;
– на какой возраст выглядит.
Класс должен обеспечивать функциональность:
– change_rating(value) – изменять рейтинг на значение value; не может стать больше 100 и меньше 1, изменяется только до достижения экстремального значения; при увеличении

рейтинга уменьшается возраст на abs(value) // 10, но только до 18, дальше не уменьшается; при уменьшении рейтинга возраст соответственно увеличивается;
– к экземпляру класса можно прибавить строку: (wd += string), значение рейтинга увеличивается на ее длину, а возраст, соответственно, уменьшается на длину // 10, условия изменения такие же;
– экземпляр класса можно вызвать с аргументом-числом; возвращает значение: (аргумент - возраст) * рейтинг;
__str__() – возвращает строку:
“Wizard <name> with <rating> rating looks <age> years old”
– экземпляры класса можно сравнивать: сначала по рейтингу, затем по возрасту, затем по имени по алфавиту; для этого нужно реализовать методы сравнения: <, >, <=, >=, ==, !=.

(14) Класс «Сотрудник компании» Worker

Экземпляр класса при инициализации принимает аргументы:
имя, должность и стаж работы сотрудника,
метод print_info() выводит информацию о сотруднике в формате:
«Имя: Василий
Должность: Системный администратор
Стаж: 3 года»
При выводе стажа нужно учитывать, что «года» должно заменяться на «лет» или «год» в зависимости от числа.

```
worker1 = Worker("Алексей", "Программист", 17)
worker1.print_info()
print()
worker2 = Worker("Анна", "Маркетолог", 2)
worker2.print_info()
print()
worker3 = Worker("Дмитрий", "Аналитик", 1)
worker3.print_info()
print()
```

(15) Класс Post

Класс описывает публикацию от пользователя в сети:
– никнейм пользователя,
– время публикации,
– количество лайков,
– текст сообщения,
– список комментариев.

Конструктор класса получает автора, устанавливает время, зануляет количество ругательств, а для комментариев создает списочный массив.

Добавить метод, позволяющий поставить лайк сообщению.

(16) Классы «Товар» и «Склад»

Класс «Товар»содержит следующие закрытые поля:
– название товара,
– название магазина в котором продается товар
– стоимость товара в рублях

Класс «Склад» содержит закрытый массив товаров.
Обеспечить следующие возможности:
– вывод информации о товаре по номеру с помощью индекса
– вывод информации о товаре, название которого введено с клавиатуры
– сортировку товаров по названию магазина, по наименованию и цене;
– перегруженную операцию сложения товаров, выполняющую сложение их цен.

(17) Классы «Клиент»(Client) и «Банк» (Bank)

Класс «Клиент» содержит поля: код клиента, ФИО, дата открытия вклада, размер вклада, процент по вкладу.

Класс «Банк» (class Bank) содержит поле clientBase представляющем собой список клиентов и методами:
– addClient(client) — принимает обьект клиента и помещает его в base.
– showByMoney(money) — принимает количество денег и выводит информацию о всех клиентах у которых размер вклада больше
– showByCode(cod) — принимает код и выводит всю информацию клиенте с данным кодом.
– showByProc(proc) — принимает процент и выводит информацию о всех клиентах у которых процент по вкладу больше данного.

(18) Класс «Автомобиль» и дочерний класс «Автобус»

Экземпляр класса имеет координаты своего положения и угол, описывающий направление движения. Он может быть изначально поставлен в любую точку с любым направлением (конструктор), может проехать в выбранном направлении определённое расстояние и может повернуть, то есть изменить текущее направление на любое другое

Реализуйте класс автомобиля, а также класс, который будет описывать автобус. Кроме того, что имеется у автомобиля, у автобуса должны быть поля, содержащие число пассажиров и количество полученных денег, изначально равные нулю. Также должны быть методы «войти» и «выйти», изменяющие число пассажиров. Наконец, метод move должен быть переопределён, чтобы увеличивать количество денег в соответствии с количеством пассажиров и пройденным расстоянием.

(19) Классы «ПЕРСОНА», «АБИТУРИЕНТ», «СТУДЕНТ», «ПРЕПОДАВАТЕЛЬ»

Класс ПЕРСОНА, экземпляр класса инициализируется аргументами фамилия, дата рождения и содержит методы, позволяющие вывести информацию о персоне, а также определить ее возраст.

Дочерние классы: АБИТУРИЕНТ (фамилия, дата рождения, факультет), СТУДЕНТ(фамилия, дата рождения, факультет, курс), ПРЕПОДАВАТЕЛЬ (фамилия, дата рождения, факультет, должность, стаж), содержат свои методы вывода информации.

Создайте список из n персон, выведите полную информацию из базы, а также организуйте поиск персон, чей возраст попадает в заданный диапазон.

22. Литература

- Бизли Д., Python, подробный справочник, 4-е издание, 2010.
- Марк Лутц, Изучаем Python, Т. 1, 5-е издание, 2019.
- Марк Лутц, Изучаем Python, Т. 2, 5-е издание, 2020.
- Марк Лутц, Программирование на Python, 4-е издание, I том, 2011.
- Марк Саммерфилд, Программирование на Python 3. Подробное руководство, 2009.
- Майкл Доусон. Программируем на Python. 3-е издание, 2014.
- Swaroop Chitlur, A Byte of Python, 2020.

Сайты:

- http://python.org/ Официальный сайт
- http://python.ru/
- http://python.su/
- http://programarcadegames.com Доходчиво описан язык на основе аркадных игр.
- http://younglinux.info/ Основы программирования на Python. Курс по информатике, ООП, tkinter, алгоритмы, решение задач.
- http://pythonworld.ru/karta-sajta Язык программирования Python 3 для начинающих и чайников.
- http://aliev.me/runestone/ Решение проблем с использованием алгоритмов и структур данных
- https://www.programiz.com/python-programming очень полезный ресурс для начинающих, все очень разжевано и с примерами, но на английском языке.
- https://www.youtube.com/channel/UCMcC_43zGHttf9bY-xJOTwA хороший YouTube канал для начинающих изучать Python.

Онлайн IDE/VM

- http://runnable.com/new/Python
- https://koding.com/IDE
- https://c9.io/

Ресурсы о популярных дополнениях

IPython - улучшенная интерактивная оболочка

- Домашняя страница: http://ipython.org/
- Быстрый старт - Introducing IPython
- Распечатай и положи на рабочий стол - IPython Quick Reference
- Полная официальная документация

matplotlib - построение графиков в различных форматах в стиле, навеянном MATLAB

- Домашняя страница: http://matplotlib.org/
- Официальная документация
- Огромный каталог примеров
- Глава в книге The Architecture of Open Source Applications от создателей matplotlib

numpy - библиотека для научных расчётов

- Домашняя страница: http://www.numpy.org/
- Пособие для начинающих
- Полная официальная документация

https://docs.python.org/3/py-modindex.html - перечень модулей с описанием функций.