top

КОМПЬЮТЕРНАЯ АКАДЕМИЯ

Основы программирования на языке Python

Урок 9

Объектно-ориентированное программирование

Contents

Основы ООП

Основы объектно-ориентированного подхода

Давайте отойдем от компьютерного программирования и компьютеров в целом и обсудим вопросы объектного программирования.

Почти все программы и методы, которые вы использовали до сих пор, подпадают под процедурный стиль программирования. Конечно, вы пользовались и кое-какими встроенными объектами, но это был абсолютный минимум.

Десятилетиями доминирующим подходом к разработке программного обеспечения был именно процедурный стиль программирования, и его используют до сих пор. Более того, он не исчезнет и в будущем, так как очень хорошо подходит для определенных проектов (как правило, не очень сложных и небольших, но из этого правила есть много исключений).

Объектный подход довольно новый (он намного моложе процедурного) и особенно полезен в больших и сложных проектах, над которыми работают большие команды разработчиков.

Такое понимание структуры проекта облегчает многие важные задачи, например, позволяет делить проект на небольшие независимые части и разрабатывать разные элементы проекта независимо друг от друга.

Python — это универсальный инструмент как для объектного, так и для процедурного программирования. Он успешно используется в обоих стилях.

Кроме того, вы можете создавать множество полезных приложений, даже если ничего не знаете о классах и объектах, но не забывайте, что определенные задачи (например, обработка графического пользовательского интерфейса) могут потребовать строгого объектного подхода.

К счастью, объектное программирование относительно простое.

Рисунок 1

Процедурный и объектно-ориентированный подход

В рамках *процедурного подхода* можно выделить два совершенно разных мира: *мир данных и мир кода*. Мир данных заполнен разными переменными, в то время как мир кода населен кодом, который сгруппирован по модулям и функциям.

Функции используют данные, но не наоборот. Кроме того, функции могут и злоупотреблять данными, то есть использовать значение несанкционированным образом (например, когда функция синуса получает баланс банковского счета в качестве параметра).

Ранее мы говорили, что данные не могут использовать функции. Но так ли это? Существуют ли какие-то особые виды данных, которые могут использовать функции?

Да, существуют и называются они «методы». Это функции, которые вызываются из данных, а не рядом с ними. Если вы видите эту разницу, то вы уже сделали первый шаг в объектное программирование.

Объектный подход предлагает совершенно другой способ мышления. Данные и код объединены в одном мире, который разделен на классы. Каждый класс похож на рецепт, который используется для создания полезного объекта (отсюда и название подхода). Вы можете создать столько объектов, сколько нужно для решения проблемы.

У каждого объекта есть набор характеристик (они называются свойствами или атрибутами — мы будем использовать оба слова как синонимы), и каждый объект может выполнять определенный набор действий (которые называются методами).

Рецепты можно менять, если они не подходят для конкретных целей и, как следствие, создавать новые классы. Эти новые классы наследуют от оригиналов их свойства и методы и обычно добавляют свои, создавая новые, более точные инструменты.

Объекты являются воплощениями идей, выраженных классами, например, чизкейк на тарелке — воплощение идеи, выраженной в рецепте, который напечатан в старой кулинарной книге.

Объекты взаимодействуют друг с другом, обмениваясь данными или активируя свои методы. Правильно построенный класс (и, следовательно, его объекты) способен защитить личную информацию и спрятать ее от несанкционированных изменений.

Нет четкой границы между данными и кодом: они живут как единое целое в объектах.

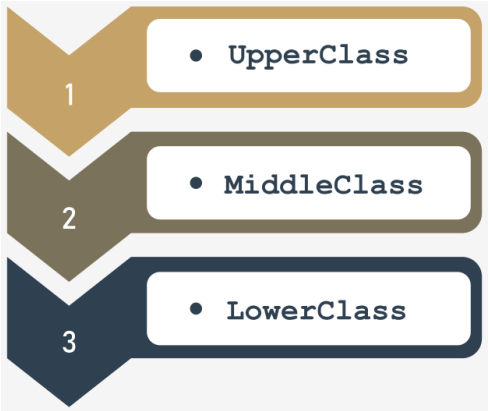

Рисунок 2

Все эти понятия не так абстрактны, как может показаться. Напротив, все они взяты из опыта реальной

жизни и поэтому чрезвычайно полезны в компьютерном программировании: они не создают искусственную жизнь — *они отражают реальные факты, отношения и обстоятельства*.

Иерархии классов

У слова класс много значений, но не все они совместимы с идеями, которые мы собираемся раскрыть в этих уроках. Интересующий нас класс похож на категорию, которая возникла в результате точно описанного сходства.

Мы попытаемся показать вам несколько классов, которые служат хорошими примерами этой концепции.

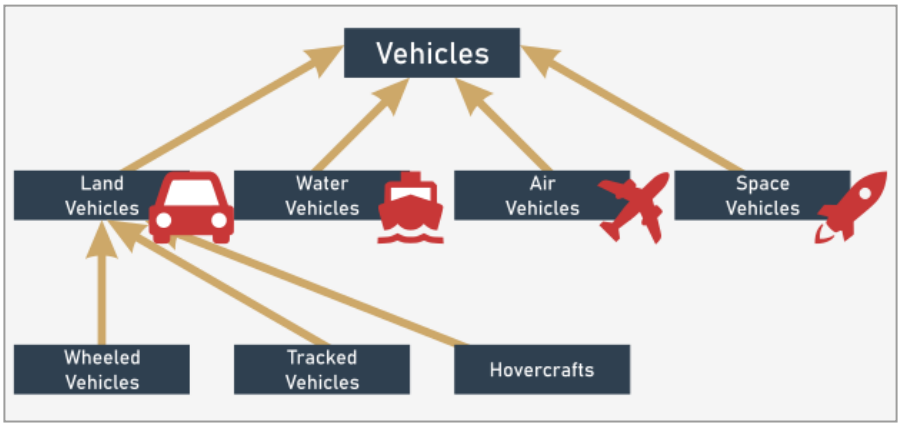

Рисунок 3

Рассмотрим транспортные средства. Все существующие транспортные средства (и те, которых еще не изобрели) связаны одной важной особенностью — способностью передвигаться. Вы можете с этим поспорить, ведь собака тоже передвигается; так что же, собака — это транспорт? Нет, конечно. Нам нужно поработать над опреде-

лением, т.е. обогатить его другими критериями, которые отличают транспорт от других существ и создают более тесную связь. Давайте рассмотрим такое определение: транспорт — это искусственно созданные объекты, используемые для транспортировки, которые движутся за счет сил природы и управляются людьми.

Согласно этому определению, собака не является транспортом.

Класс транспорт очень широкий. Слишком широкий. Это значит, что теперь нам надо дать определение еще и специализированным классам. Специализированные классы — это подклассы. Класс транспорт будет суперклассом для них всех.

Примечание: иерархия растет сверху вниз, как корни деревьев, а не ветви. Самый общий и самый широкий класс всегда находится вверху (суперкласс), в то время как его потомки расположены ниже (подклассы).

Вы, скорее всего, уже можете выделить некоторые возможные подклассы суперкласса транспорт. Есть много разных классификаций. Мы выбрали разделение на подклассы на основе среды и говорим, что есть (как минимум) четыре подкласса:

- наземный транспорт;
- водный транспорт;
- воздушный транспорт;
- космический транспорт,

В этом примере мы обсудим только первый подкласс — наземный транспорт. Если хотите, вы можете самостоятельно разобрать остальные классы.

Наземный транспорт можно далее поделить на группы в зависимости от того, как они соприкасаются с землей. Итак, можно выделить:

- колесный транспорт;
- транспорт на гусеничном ходу;
- на воздушной подушке.

На рисунке выше представлена иерархия, которую мы создали. Обратите внимание на направление стрелок — они всегда указывают на суперкласс. Класс верхнего уровня является исключением — у него нет своего суперкласса.

Еще один пример — иерархия царств животных. Можно смело разделить всех животных (наш класс верхнего уровня) на пять подклассов:

- млекопитающие;
- рептилии;
- птицы;
- рыбы;
- амфибии.

Для дальнейшего анализа возьмем первый.

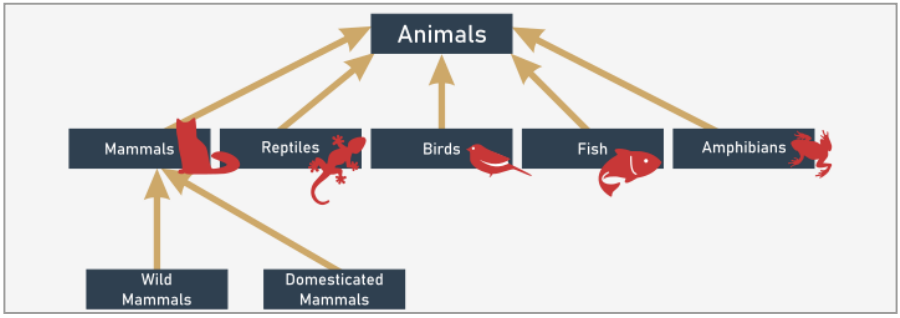

Рисунок 4

Мы выделили следующие подклассы:

- дикие млекопитающие;
- одомашненные млекопитающие.

Попробуйте расширить иерархию так, как вы хотите, и найдите подходящее место для людей (рис. 4).

Что такое объект?

Класс (вдобавок к другим определениям) является набором объектов. Объект — это нечто, принадлежащее классу.

Объект — это воплощение требований, характеристик и качеств, которые приписываются конкретному классу. Это может показаться простым, но обратите внимание на следующие важные особенности. Классы образуют иерархию. Это означает, что объект, принадлежащий определенному классу, одновременно принадлежит всем суперклассам. Это также означает, что любой объект, принадлежащий суперклассу, может не принадлежать ни одному из его подклассов.

Например: любой личный автомобиль — это объект, принадлежащий классу колесный транспорт. Это также означает, что один и тот же автомобиль принадлежит всем суперклассам своего класса; следовательно, он также является членом класса транспорт. Ваша собака (или ваша кошка) является объектом, который входит в класс одомашненные млекопитающие, а это прямо указывает на то, что он также принадлежит классу животные.

Каждый подкласс всегда уже (или конкретнее), чем его суперкласс. И наоборот, каждый суперкласс всег-

да более общий (более абстрактный), чем любой из его подклассов. Обратите внимание, мы сказали, что у класса может быть только один суперкласс, но это не всегда так. Мы обсудим эту проблему чуть позже.

Наследование

Давайте дадим определение одной из фундаментальных концепций объектного программирования, которая называется *наследование*. Любой объект, связанный с определенным уровнем иерархии классов, *наследует все характеристики (а также требования и качества), описанные во всех его суперклассах*.

Родной класс объекта может описывать новые характеристики (а также требования и качества), которые будут наследоваться любым из его суперклассов.

У вас не должно возникнуть проблем с поиском конкретных примеров для этого правила, будь то животные или транспорт.

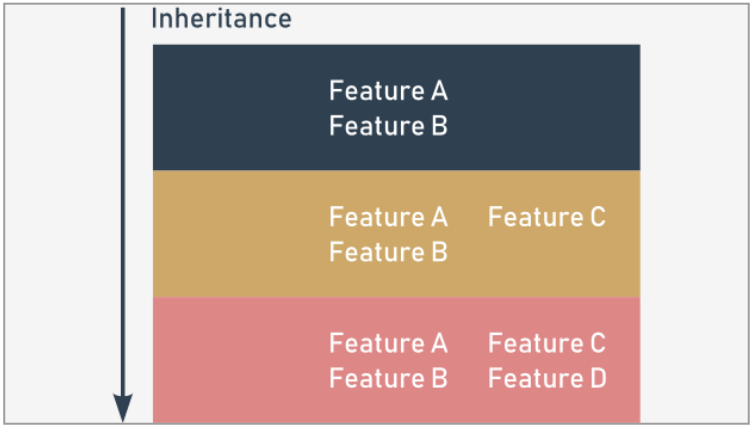

Рисунок 5

Что есть у объекта?

Соглашение об объектном программировании предполагает, что *любой существующий объект обладает тремя группами атрибутов*:

- у объекта есть *имя*, которое однозначно идентифицирует его в своем родном пространстве имен (хотя анонимные объекты тоже встречаются);
- у объекта есть *набор индивидуальных свойств*, которые делают его оригинальным, уникальным или особенным (хотя у некоторых объектов может вообще не быть свойств);
- у объекта есть *набор способностей для выполнения определенных действий*, которые могут изменять сам объект или какие-то другие объекты.

Есть подсказка (хотя она не всегда работает), которая может помочь вам распознавать вышеназванные атрибуты. Если при описании объекта вы используете:

- *существительное* — скорее всего, вы определяете имя объекта;
- *прилагательное* — скорее всего, вы определяете свойство объекта;
- *глагол* — скорее всего, вы определяете действие объекта.

Ниже даны два примера, которые наглядно демонстрируют, как это работает:

- **Макс — большой кот, который целый день спит.**
  - Имя объекта = Макс
  - Класс = Кот
  - Свойство = Размер (большой)
  - Действие = Сон (целый день).

- **Розовый кадиллак быстро уехал.**
  - Имя объекта = Кадиллак
  - Класс = Колесный транспорт
  - Свойство = Цвет (розовый)
  - Действие = Ехать (быстро).

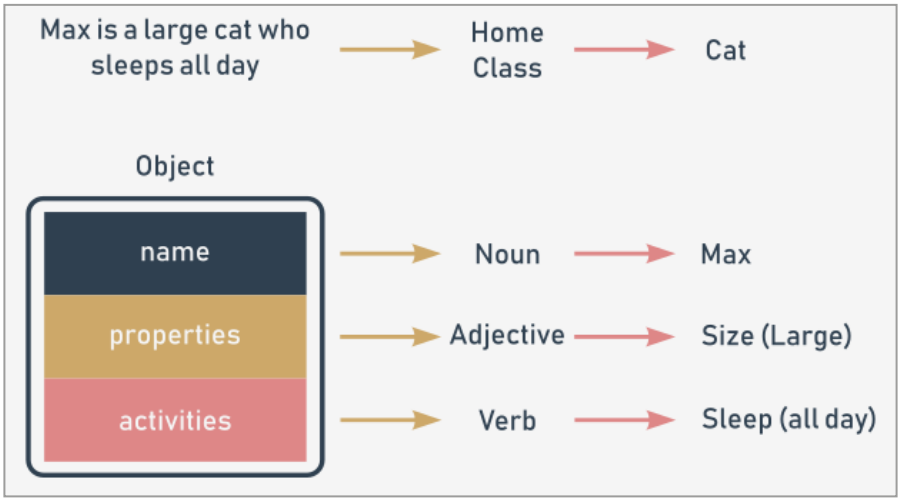

Рисунок 6

Пишем первый класс

Объектное программирование это искусство определять и расширять классы. **Класс** — это модель особой стороны реальности, которая отражает свойства и действия, встречающиеся в реальном мире.

Классы, которые мы обсуждали в начале, слишком общие и неточные, чтобы охватить наибольшее количество реальных объектов.

Но вы всегда можете определить новые, более точные подклассы. Они унаследуют все от своего суперкласса,

поэтому силы, потраченные на его создание, не пропадут даром.

Новый класс может добавлять новые свойства и новые действия, и, следовательно, может быть более полезным в конкретных ситуациях. Очевидно, что его можно использовать в качестве суперкласса для любого количества создаваемых подклассов.

Этот процесс может стать бесконечным. Вы можете создать столько классов, сколько нужно.

Определенный вами класс не имеет ничего общего с объектом: *наличие класса не означает, что любой подходящий объектов создастся автоматически*. Сам класс не может создавать объект — вам придется создать его самостоятельно, и Python дает вам в руки все необходимые инструменты.

Пришло время определить самый простой класс и создать объект. Посмотрите на пример ниже:

```
class TheSimplestClass:
      pass
```

Здесь мы определили класс. Класс довольно бедный: у него нет ни свойств, ни действия. Вообще-то, он пустой, но пока что это неважно. Чем проще класс, тем для наших целей лучше.

Определение начинается с ключевого слова class. За ключевым словом следует идентификатор, который назовет класс.

> **Примечание:** *не путайте его с именем объекта — это две разные вещи).*

Далее вы добавляете двоеточие «:», потому что классы, как и функции, образуют собственный вложенный блок. Содержимое внутри блока определяет все свойства и действия класса.

Ключевое слово pass ничем не заполняет класс. Оно не содержит никаких методов или свойств.

Пишем первый объект

Только что определенный класс становится инструментом, с помощью которого мы будем создавать новые объекты. Инструмент должен использоваться явно, по требованию.

Представьте, что вы хотите создать один (ровно один) объект класса TheSimplestClass.

Для этого вам нужно назначить переменную для хранения созданного объекта этого класса и одновременно создать объект.

Это делается следующим образом:

```
myFirstObject = TheSimplestClass()
```

Примечание:

- *имя класса пытается притвориться функцией — вы это видите? Скоро мы это обсудим;*
- *созданный объект оснащен всем, что есть у класса, но так как этот класс полностью пуст, то и объект тоже пуст.*

Акт создания объекта выбранного класса также называется инстанцированием (потому что объект становится экземпляром (instance) класса).

Давайте ненадолго оставим классы и познакомимся со стеками. Мы понимаем, что понятия классов и объектов вам пока еще не до конца ясны. Не переживайте, скоро мы все очень подробно разберем.

Короткий путь от процедурного подхода к объектному

Что такое стек?

Стек — это структура, разработанная для очень специфического хранения данных. Представьте себе стопку монет. Новую монету можно положить только на верхнюю монету в стопке. Точно так же и достать из стопки можно только верхнюю монету. Если нужно достать самую нижнюю монету, сначала нужно снять все монеты над ней.

Другое название стека (но только в терминологии ИТ): *LIFO*. Это аббревиатура, которая очень точно описывает поведение стека: *Last In — First Out* (последним пришел — первым вышел). Монета, которая попала в стопку последней, уйдет первой.

Стек — это объект с двумя элементарными операциями, условно названными push (добавление или проталкивание; когда новый элемент помещается сверху) и pop (удаление, выталкивание; когда элемент убирается сверху).

Стеки используются во многих классических алгоритмах, и трудно представить реализацию многих широко используемых инструментов без использования стеков (рис. 7).

Давайте реализуем стек на языке Python. Это будет очень простой стек, и мы покажем вам, как его создать, используя два независимых подхода: процедурный и объектный.

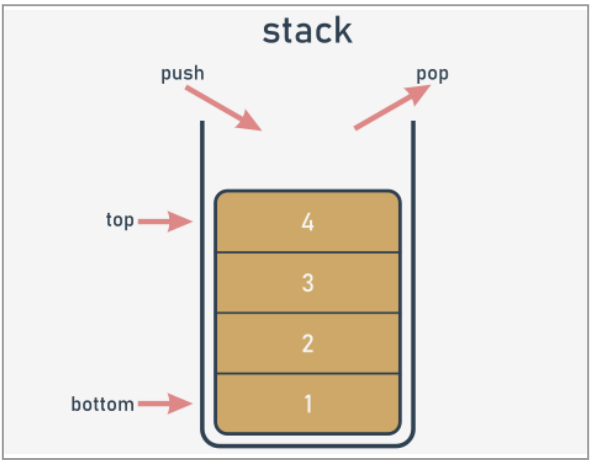

Рисунок 7

Начнем с первого.

Стек — процедурный подход

Сначала вы должны решить, как будете хранить значения, которые поступят в стек. Мы предлагаем использовать самые простые методы, и применить список для этих целей. Предположим, размер стека ничем не ограничен. Также предположим, что последний элемент списка хранит верхний элемент.

Сам стек уже создан:

```
stack = []
```

Мы готовы определить функцию, которая помещает значение в стек. Вот что у нас есть:

- имя функции — push;
- функция получает один параметр (это значение, которое попадет в стек);

- функция ничего не возвращает;
- функция добавляет значение параметра в конец стека.

Вот как мы это сделали — посмотрите:

```
def push(val):
    stack.append(val)
```

Теперь функция должна удалить значение из стека. Вот как это можно сделать:

- название функции — pop;
- функция не получает никаких параметров;
- функция возвращает значение, взятое из стека
- функция считывает значение с вершины стека и удаляет его.

Функция получается такая:

```
def pop():
    val = stack[-1]
    del stack[-1]
    return val
```

Примечание: *функция не проверяет, есть ли что-то в стеке.*

Давайте соберем все части вместе, чтобы реализовать стек. Готовая программа помещает три числа в стек, достает их и выводит их значения на экран. На рисунке ниже представлен полный код программы.

```
stack = []
def push(val):
    stack.append(val)
```

```
def pop():
    val = stack[-1]
    del stack[-1]
    return val
push(3)
push(2)
push(1)
print(pop())
print(pop())
print(pop())
```

Программа выводит на экран следующее:

```
1
2
3
```

Протестируйте ее.

Стек — процедурный и объектно-ориентированный подходы

Процедурный стек готов. Конечно, в нем есть и недостатки, и реализацию можно по-разному улучшать (использовать исключения — хорошая мысль), но в целом стек полностью реализован, и его можно использовать при необходимости.

Но чем чаще вы его используете, тем с большим количеством недостатков в итоге столкнетесь. Вот лишь некоторые из них:

- основная переменная (список стека) очень уязвима; кто угодно может изменить ее как угодно, фактически разрушив стек. И не обязательно намеренно —

наоборот, это может произойти в результате небрежности, например, если программист перепутает имена переменных. Представьте, что вы случайно написали что-то вроде этого:

```
stack[0] = 0
```

Работа стека будет нарушена;

- или вам однажды может понадобиться второй стек, тогда придется создать еще один список для хранения стека, и, возможно, еще одну пару функций push и pop;
- или представьте, что вам нужны не только функции push и pop, но и другие удобные инструменты. Вы, конечно, можете их реализовать, но попытайтесь представить, что произойдет, если у вас будут десятки отдельно реализованных стеков.

Объектный подход решает все вышеуказанные проблемы. Давайте сначала назовем эти решения:

- возможность скрывать (защищать) выбранные значения от несанкционированного доступа называется *инкапсуляцией*; *к инкапсулированным значениям нельзя получить доступ или изменить их, если использовать только их*;
- если у вас есть класс, реализующий все необходимое поведение стека, можно создать столько стеков, сколько захотите, и при этом нет необходимости копировать какую-либо часть кода;
- возможность обогащать стек новыми функциями связана с наследованием. Можно создать новый класс (подкласс), который наследует все характеристики от суперкласса и добавляет новые.

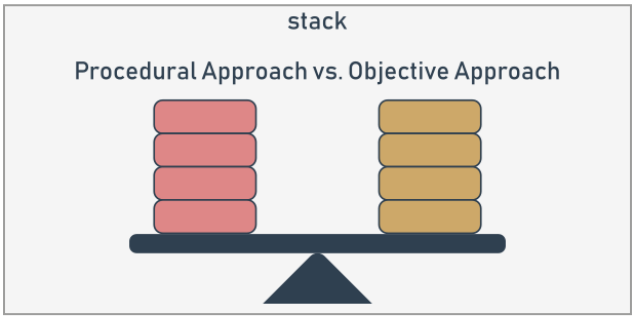

Рисунок 8

Давайте теперь напишем совершенно новую реализацию стека с нуля. На этот раз мы будем использовать объектный подход, шаг за шагом углубляясь в мир объектно-ориентированного программирования.

Стек — объектный подход

Конечно, основная идея остается прежней. Мы будем использовать список в качестве хранилища стека. Нам только нужно знать, как поместить список в класс.

Давайте начнем с самого начала — вот как начинается объектный стек:

```
class Stack:
```

Теперь нам от него нужны две вещи:

- чтобы у класса было одно свойство — хранилища стека. Надо «установить» список внутри каждого объекта класса (примечание: у каждого объекта должен быть свой собственный список — список не должен быть общим для разных стеков);
- дальше мы хотим, чтобы список был скрыт от пользователей класса.

Как это сделать?

В отличие от других языков программирования, в Python нет средств, позволяющих просто взять и объявить такое свойство.

Вместо этого надо добавить конкретное утверждение или инструкцию. Свойства добавляются в класс вручную.

А как гарантировать, что это будет происходить каждый раз при создании нового стека?

Есть простой способ — добавить в класс определенную функцию. Но есть два условия:

- строгое именование функции;
- она вызывается неявно, во время создания нового объекта.

Такая функция называется конструктор, потому что ее цель — сконструировать (создать, построить) новый объект. Конструктор должен знать все о структуре объекта и выполнять все необходимые инициализации.

Давайте добавим очень простой конструктор в новый класс. Посмотрите на этот фрагмент кода:

```
class Stack:
    def __init__(self):
        print("Hi!")
stackObject = Stack()
```

Давайте разбираться:

- имя конструктора всегда будет __init__;
- у него должен быть хотя бы один параметр (мы обсудим это позже); параметр используется для представления недавно созданного объекта — этот параметр можно

использовать для управления объектом и чтобы добавлять в него необходимые свойства;

Примечание: *обязательный параметр обычно называется self — это только общепринятое правило, но вы обязаны ему следовать — это упрощает процесс чтения и понимания вашего кода.*

Код представлен ниже.

```
class Stack: # defining the Stack class
   def __init__(self): # defining the constructor
                       # function
       print("Hi!")
stackObject = Stack()     # instantiating the object
```

Запустите его.

Вот что он выводит:

```
Hi!
```

Обратите внимание — нет никаких признаков вызова конструктора внутри кода. Его вызвали неявно и автоматически. Давайте воспользуемся этим.

Любые изменения внутри конструктора, который изменяет состояние параметра self, будут отражаться на созданном объекте.

Это означает, что в объект можно добавить любое свойство, и оно будет оставаться там до тех пор, пока объект не прекратит существовать или свойство не будет явно удалено.

Теперь давайте добавим только одно свойство в новый объект — список для стека. Назовем его «stackList».

Как здесь:

```
class Stack:
    def __init__(self):
        self.stackList = []
stackObject = Stack()
print(len(stackObject.stackList))
```

Примечание:

- *мы использовали запись с точками, как при вызове методов. Это общее правило для доступа к свойствам объекта — нужно дать имя объекту, поставить точку (.) после него и указать желаемое имя свойства. Не используйте скобки! Нам не нужно вызвать метод, нам надо получить доступ к свойству;*
- *если значение свойства устанавливается впервые (как в конструкторе), то в этот момент создается это свойство с уже присвоенным значением. После этого объект приобрел свойство и готов использовать его значение;*
- *но в коде есть кое-что еще — мы попытались получить доступ к свойству stackList вне класса сразу после создания объекта. То есть мы хотим проверить текущую длину стека — добились ли мы успеха?*

Да — код выдает следующий результат:

```
0
```

Однако это не то, чего мы хотим от стека. Нам нужно, чтобы stackList был скрыт от внешнего мира. Это возможно?

Да, и это просто, но не особенно интуитивно понятно.

Посмотрите — мы добавили два нижних подчеркивания перед именем stackList — и больше ничего:

```
class Stack:
    def __init__(self):
        self.__stackList = []

stackObject = Stack()
print(len(stackObject.__stackList))
```

Это изменение ломает программу. Почему?

Если имя компонента класса начинается с двух подчеркиваний (__), он становится приватным (private). Это означает, что к нему можно получить доступ только из класса.

Его нельзя увидеть извне. Вот так Python реализует понятие инкапсуляции.

Запустите программу, чтобы проверить наши предположения — должно появиться исключение AttributeError.

Объектный подход: стек с нуля

Теперь пришло время для двух функций (методов), реализующих операции push и pop. Python предполагает, что функция такого рода (действие класса) должна быть погружена в тело класса так же, как конструктор.

Мы хотим вызвать эти функции со значениями push и pop. Это означает, что они обе должны быть доступны любому пользователю класса (в отличие от ранее созданного списка, который скрыт от обычных пользователей класса).

Такой компонент называется публичным (public), так что здесь нельзя начинать имя с двух (или более) подчеркиваний. Есть еще одно требование — имя должно содержать не более одного подчеркивания в конце. Поскольку «ноль подчеркиваний» полностью соответствует этому требованию, можно считать, что имя приемлемо.

Сами функции просты. Посмотрите:

```
class Stack:
    def __init__(self):
        self.__stackList = []

def push(self, val):
    self.__stackList.append(val)

def pop(self):
    val = self.__stackList[-1]
    del self.__stackList[-1]
    return val

stackObject = Stack()

stackObject.push(3)
stackObject.push(2)
stackObject.push(1)

print(stackObject.pop())
print(stackObject.pop())
print(stackObject.pop())
```

Но в этом коде есть кое-что очень странное. Функции выглядят знакомо, но у них больше параметров, чем у их процедурных аналогов.

Здесь у обеих функций есть параметр с именем self в первой позиции списка параметров. Он нужен? Да.

У всех методов должен быть этот параметр. Он играет ту же роль, что и первый параметр конструктора.

Он позволяет методу получать доступ к объектам (свойствам и действиям/методам), которые выполняются фактическим объектом. Им нельзя пренебрегать. Каждый раз, когда Python вызывает метод, он неявно отправляет текущий объект в качестве первого аргумента.

Это означает, что метод обязан иметь хотя бы один параметр, который использует сам Python — вы никак не можете на это повлиять.

Даже если вашему методу не нужны никакие параметры, этот параметр должен быть указан в любом случае. Если его задача — обработать только один параметр, нужно указывать два. Роль первого из них остается прежней.

Есть еще одна особенность, которая требует объяснения — способ вызова методов из переменной __stackList.

К счастью, все гораздо проще, чем кажется:

- на первом этапе объект передается как одно целое → self;
- далее нужно добраться до списка __stackList → self.__stackList;
- теперь, когда __stackList готов к использованию, можно переходить к третьему и последнему этапу → self.__stackList.append(val).

Объявление класса закончено, и все его компоненты перечислены. Класс готов к использованию.

Наличие такого класса открывает новые возможности. К примеру, теперь у вас может быть больше одного стека, а вести себя они будут одинаково. У каждого сте-

ка будет собственная копия приватных данных, но набор методов останется неизменным.

Это именно то, что нам нужно для этого примера.

Проанализируйте следующий код:

```
class Stack:
    def __init__(self):
        self.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
        del self.__stackList[-1]
        return val

stackObject1 = Stack()
stackObject2 = Stack()

stackObject1.push(3)
stackObject2.push(stackObject1.pop())

print(stackObject2.pop())
```

Есть два стека, которые созданы из одного базового класса. Они работают независимо друг от дргуа. Но их может быть и больше, чем два.

Запустите код в редакторе и посмотрите, что получится. Проведите свои собственные эксперименты.

Проанализируйте фрагмент кода ниже — мы создали три объекта класса Stack. Потом мы ими слегка пожонглировали. Попробуйте предсказать значение, которое выведется на экран.

```
class Stack:
    def __init__(self):
        self.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
        del self.__stackList[-1]
        return val

littleStack = Stack()
anotherStack = Stack()
funnyStack = Stack()

littleStack.push(1)
anotherStack.push(littleStack.pop() + 1)
funnyStack.push(anotherStack.pop() — 2)

print(funnyStack.pop())
```

Итак, каким будет результат? Запустите программу и проверьте, были ли вы правы.

Теперь давайте пойдем немного дальше. Давайте добавим новый класс для обработки стеков.

```
class Stack:
    def __init__(self):
        self.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
```

```
        del self.__stackList[-1]
        return val

class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0
```

Новый класс должен посчитать сумму всех элементов, хранящихся в данный момент в стеке.

Мы не хотим менять ранее определенный стек. Он уже достаточно хорошо справляется со своей задачей, поэтому не стоит там ничего менять. Нам нужен новый стек, с новыми возможностями. Другими словами, мы хотим построить подкласс уже существующего класса Stack.

Первый шаг прост: нужно определить новый подкласс, указывающий на класс, который будет использоваться в качестве суперкласса.

Вот как это выглядит:

```
class AddingStack(Stack):
pass
```

Класс еще не определил новый компонент, но это не значит, что он пуст. Он получает все компоненты, которые были определены его суперклассом — имя суперкласса пишется после двоеточия сразу после нового имени класса.

Вот что мы хотим от нового стека:

- чтобы метод push не только помещал значение в стек, но и добавлял значение в переменную sum;

- чтобы функция pop не только извлекала значение из стека, но и вычитала значение из переменной sum.

Для начала, давайте добавим новую переменную в класс. Это будет приватная переменная, как список стека. Мы не хотим, чтобы у кого-то был доступ к значению sum.

Как вы уже знаете, именно конструктор добавляет новое свойство в класс. А еще вы знаете, как это сделать, но внутри конструктора есть нечто любопытное. Посмотрите:

```
class AddingStack(Stack):
def __init__(self):
Stack.__init__(self)
self.__sum = 0
```

Вторая строка тела конструктора создает свойство с именем __sum. Оно будет хранить сумму всех значений стека.

Но строка выше выглядит иначе. Что она делает? Она действительно необходима? Да.

В отличие от многих других языков, Python требует явного вызова конструктора суперкласса. Если вы это упустите, последствия будут не самыми приятными — объект потеряет список __stackList. Такой стек не будет нормально работать.

Это единственный раз, когда можно явно вызвать любой из доступных конструкторов. Это делается внутри конструктора суперкласса.

Обратите внимание на синтаксис:

- сначала нужно указать имя суперкласса (это класс, конструктор которого вы хотите запустить);

- поставить точку (.) после него;
- указать имя конструктора;
- указать на объект (экземпляр класса), который должен быть инициализирован конструктором — вот почему здесь обязательно нужно указать аргумент и использовать переменную self.

Примечание: *вызов метода (включая конструкторы) вне класса никогда не потребует от вас аргумента self в списке аргументов. А вызов метода из класса требует явного использования аргумента self, и он должен стоять на первом месте в списке.*

Примечание: *как правило, рекомендуется вызывать конструктор суперкласса перед любыми другими инициализациями, которые выполняются внутри подкласса. Это правило, которому мы следовали в этом фрагменте кода.*

Теперь давайте добавим два метода. Но позвольте спросить: разве мы на самом деле будем их добавлять? У нас уже есть эти методы в суперклассе. Можем ли мы сделать что-то похожее?

Да, можем. Это означает, что мы будем менять функционал методов, а не их имена, Если точнее, интерфейс (способ обработки объектов) класса остается неизменным при одновременном изменении реализации.

Давайте начнем с реализации функции push. Вот что нам от нее нужно:

- добавить значение к переменной __sum;
- поместить значение в стек.

Примечание: *второе действие уже реализовано внутри суперкласса, так что используем его. Более того, мы обязаны его использовать, потому что другого способа получить доступ к переменной __stackList нет.*

Вот как метод push выглядит в подклассе:

```
def push(self, val):
    self.__sum += val
    Stack.push(self, val)
```

Обратите внимание на то, как мы вызывали предыдущую реализацию метода push (ту, которая доступна в суперклассе):

- мы должны указать имя суперкласса; это необходимо для того, чтобы четко указать класс, содержащий метод, чтобы не путать его с другой функцией с тем же именем;
- мы должны указать целевой объект и передать его в качестве первого аргумента (он неявно добавляется к вызову в этом случае).

Мы говорим, что метод push был переопределен — то же имя, что и в суперклассе, теперь представляет другой функционал.

Это новая функция pop:

```
def pop(self):
    val = Stack.pop(self)
    self.__sum -= val
    return val
```

Пока что мы определили переменную __sum, но не передали метод, который бы получил ее значение. Кажется, он скрыт. Как мы можем его обнаружить и сделать так, чтобы он по-прежнему защищал ее от изменений?

Нужно назначить новый метод. Назовем его getSum. Его единственной задачей будет вернуть значение __sum.

Вот так:

```
def getSum(self):
return self.__sum
```

Итак, посмотрите на программу ниже.

```
class Stack:
    def __init__(self):
        self.__stackList = []

    def push(self, val):
        self.__stackList.append(val)

    def pop(self):
        val = self.__stackList[-1]
        del self.__stackList[-1]
        return val

class AddingStack(Stack):
    def __init__(self):
        Stack.__init__(self)
        self.__sum = 0

    def getSum(self):
        return self.__sum

    def push(self, val):
        self.__sum += val
        Stack.push(self, val)
```

```
    def pop(self):
        val = Stack.pop(self)
        self.__sum -= val
        return val

stackObject = AddingStack()

for i in range(5):
    stackObject.push(i)
print(stackObject.getSum())

for i in range(5):
    print(stackObject.pop())
```

Она содержит полный код класса. Сейчас можно проверить, как он работает. И мы это сделали с помощью нескольких дополнительных строк кода.

Как видите, мы добавили в стек пять следующих значений, вывели их сумму и убрали их все из стека.

Хорошо, это было очень краткое введение в объектно-ориентированное программирование на Python. Вскоре мы расскажем вам обо всем подробнее.

ООП: Свойства

Переменные экземпляра

Как правило, у класса есть два разных типа данных, которые формируют свойства класса. Вы уже знаете об одном из них — мы с ним познакомились, когда разбирали стеки.

Такое свойство класса существует только при условии, что оно было явно создано и добавлено в объект. Свойство класса можно создать и добавить в объект не только во время его инициализации (как мы делали ранее), но вообще в любой момент жизни объекта. Кроме того, любое свойство и удалить можно в любой момент.

У такого подхода есть несколько важных последствий:

- разные объекты одного и того же класса могут обладать разными наборами свойств;
- должен быть способ безопасно проверить, принадлежит ли используемое свойство конкретному объекту (об этом всегда нужно помнить, чтобы избежать исключений)
- каждый объект несет свой собственный набор свойств, и они никак не мешают друг другу.

Такие переменные (свойства) называются переменными экземпляра.

Слово «экземпляр» намекает, что они тесно связаны с объектами (которые являются экземплярами классов), а не с самими классами. Давайте рассмотрим их подробнее на этом примере:

```
class ExampleClass:
    def __init__(self, val = 1):
        self.first = val
    def setSecond(self, val):
        self.second = val

exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)

exampleObject2.setSecond(3)

exampleObject3 = ExampleClass(4)
exampleObject3.third = 5

print(exampleObject1.__dict__)
print(exampleObject2.__dict__)
print(exampleObject3.__dict__)
```

Здесь требуется одно дополнительное объяснение, прежде чем мы углубимся в детали. Посмотрите на последние три строки кода.

При создании объекты Python наделяются небольшим набором заранее определенных свойств и методов. Они есть у каждого объекта, хотите вы того или нет. Одним из таких свойств является переменная с именем __dict__ (сокращенно от «*dictionary*», словарь).

Эта переменная содержит имена и значения всех свойств (переменных), которые в данный момент есть у объекта. Мы будем ее использовать для безопасного представления содержимого объекта.

А сейчас углубимся в код:

- У класса под названием «ExampleClass» есть конструктор, который в обязательном порядке создает пере-

менную экземпляра с именем «first» и устанавливает ее значение, которое передается через первый аргумент (с точки зрения пользователя класса) или второй аргумент (с точки зрения конструктора). Обратите внимание на значение параметра по умолчанию — все, что можно сделать с обычным параметром функции, применимо и к методам;

- у класса также есть метод, который создает еще одну переменную экземпляра с именем «second»;
- мы создали три объекта класса ExampleClass, но все эти экземпляры чем-то отличаются друг от друга:
  - у объекта exampleObject1 есть только одно свойство с именем first;
  - у объекта exampleObject2 есть два свойства: first и second;
  - объекту exampleObject3 на ходу добавили свойство с именем third, хотя мы это сделали вне кода класса, это возможно и вполне допустимо.

Результат вывода программы ясно показывает, что наши предположения верны. Вот что она выводит:

```
{'first': 1} {'second': 3, 'first': 2}
             {'third': 5, 'first': 4}
```

Из этого следует еще один вывод: изменение переменной экземпляра объекта не влияет на все остальные объекты. Переменные экземпляра идеально изолированы друг от друга.

Посмотрите на измененный пример ниже.

```
class ExampleClass:
    def __init__(self, val = 1):
        self.__first = val

    def setSecond(self, val = 2):
        self.__second = val


exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)

exampleObject2.setSecond(3)

exampleObject3 = ExampleClass(4)
exampleObject3.__third = 5


print(exampleObject1.__dict__)
print(exampleObject2.__dict__)
print(exampleObject3.__dict__)
```

Он почти такой же, как предыдущий. Разница только в именах свойств. Мы добавили двойное подчеркивание (__) перед ними.

Как вы знаете, такое подчеркивание превращает переменную экземпляра из публичной в приватную, недоступную извне.

Реальное поведение этих имен немного сложнее, поэтому давайте запустим программу. Вот то, что выводится на экран:

```
{'_ExampleClass__first': 1}
{'_ExampleClass__first': 2, '_ExampleClass__second': 3}
{'_ExampleClass__first': 4, '__third': 5}
```

Вы ведь видите эти странные имена с подчеркиваниями? Как они появились?

Когда Python видит, что вы хотите добавить в объект переменную экземпляра, и это происходит внутри какого-нибудь метода объекта, он вмешивается в операцию следующим образом:

- ставит имя класса перед тем, которое написали вы;
- добавляет еще одно подчеркивание в начале.

Вот почему __first становится _ExampleClass__first.

Имя теперь полностью доступно вне класса. Запустите нижеприведенный код следующим образом:

```
print(exampleObject1._ExampleClass__first)
```

и вы получите ожидаемый результат, без ошибок и исключений.

Как видите, манипулирование приватностью свойства имеет свои ограничения.

Python не станет вмешиваться в код, если переменную экземпляра добавить вне кода класса. Тогда такое свойство будет вести себя так же, как и любое другое свойство.

Переменные класса

Переменная класса — это свойство, которое существует только в единственном экземпляре и хранится вне объекта.

> **Примечание:** *переменной экземпляра не существует, если в классе нет объекта. А переменная класса существует в единственном экземпляре, даже если в классе нет объектов.*

Создание переменных класса отличается от создания экземпляров. Пример ниже даст вам больше информации:

```
class ExampleClass:
    counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.counter += 1

exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)
exampleObject3 = ExampleClass(4)

print(exampleObject1.__dict__, exampleObject1.counter)
print(exampleObject2.__dict__, exampleObject2.counter)
print(exampleObject3.__dict__, exampleObject3.counter)
```

Смотрите:

- в первом списке определения класса есть присваивание — оно присваивает переменной counter значение 0; инициализация переменной внутри класса, но вне метода этого класса, превращает переменную в переменную класса;
- доступ к такой переменной выглядит так же, как доступ к любому атрибуту экземпляра — он находится в теле конструктора; как видите, конструктор увеличивает переменную на единицу; по сути, переменная считает все созданные объекты.

Этот код выдает следующий результат:

```
{'_ExampleClass__first': 1} 3
{'_ExampleClass__first': 2} 3
{'_ExampleClass__first': 4} 3
```

Из этого примера вытекают два важных вывода:

- переменные класса не отображаются в объекте __dict__ (это нормально, поскольку переменные класса не являются частями объекта), но всегда можно попытаться найти переменную с тем же именем, но на уровне класса — мы скоро разберем, как это делается;
- переменная класса всегда представляет одно и то же значение во всех экземплярах (объектах) класса.

Изменение имени переменной класса приведет к уже знакомому нам эффекту.

Посмотрите на пример ниже.

```
class ExampleClass:
    __counter = 0
    def __init__(self, val = 1):
        self.__first = val
        ExampleClass.__counter += 1

exampleObject1 = ExampleClass()
exampleObject2 = ExampleClass(2)
exampleObject3 = ExampleClass(4)


print(exampleObject1.__dict__,
      exampleObject1._ExampleClass__counter)
print(exampleObject2.__dict__,
      exampleObject2._ExampleClass__counter)
print(exampleObject3.__dict__,
      exampleObject3._ExampleClass__counter)
```

Вы сможете предсказать результат?

Запустите программу и проверьте правильность своих прогнозов. Все работает как положено, верно?

Мы уже упоминали, что переменные класса существуют, даже если экземпляр (объект) класса не был создан.

Теперь мы хотим воспользоваться этой возможностью, чтобы показать вам, разницу между двумя переменными __dict__, той, что из класса и той, что из объекта.

Посмотрите на код.

```
class ExampleClass:
    varia = 1
    def __init__(self, val):
        ExampleClass.varia = val

print(ExampleClass.__dict__)
exampleObject = ExampleClass(2)

print(ExampleClass.__dict__)
print(exampleObject.__dict__)
```

Проанализируем его внимательно:

- в коде определяется один класс с именем ExampleClass;
- класс определяет одну переменную класса с именем varia;
- конструктор класса устанавливает переменную со значением параметра;
- именование переменной — самая важная часть этого примера, потому что:
  - если изменить присваивание на self.varia = val, то будет создана переменная экземпляра с тем же именем, что и у класса;
  - если изменить присваивание на varia = val, то это повлияет на локальную переменную метода (мы

настоятельно рекомендуем вам протестировать обе вышеупомянутые ситуации — это поможет вам запомнить разницу);

- первая строка кода, который вне класса, выведет на экран значение атрибута ExampleClass.varia (значение используется до того, как будет создан самый первый объект класса).

Запустите код в редакторе и проверьте результат.

Как видите, класс __dict__ содержит гораздо больше данных, чем одноименный объект. Большая часть этих данных нам сейчас не нужна, а та, на которую мы хотим обратить ваше внимание, показывает текущее значение varia.

Обратите внимание, что __dict__ объекта пустой — у объекта нет переменных экземпляра.

Проверка существования атрибута

Инстанциация объектов в Python поднимает одну важную проблему — в отличие от других языков программирования, в Python не все объекты одного класса имеют одинаковый набор свойств.

Вот как в этом примере. Внимательно посмотрите на него.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1
```

```
exampleObject = ExampleClass(1)

print(exampleObject.a)
print(exampleObject.b)
```

Объект, созданный конструктором, может иметь только один из двух возможных атрибутов: a или b.

Выполнение кода даст следующий результат:

```
1
Traceback (most recent call last):
File ".main.py", line 11, in
print(exampleObject.b)
AttributeError: 'ExampleClass' object has no attribute 'b'
```

Как видите, доступ к несуществующему атрибуту объекта (класса) приводит к исключению AttributeError.

Инструкция try-except позволяет избежать проблем с несуществующими свойствами.

Это просто — посмотрите на код.

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1
exampleObject = ExampleClass(1)
print(exampleObject.a)

try:
    print(exampleObject.b)
except AttributeError:
    pass
```

Как видите, это не очень сложно. Но, на самом деле, мы только что просто отмахнулись от проблемы.

К счастью, есть один действенный способ.

У Python есть функция, которая может безопасно проверять, содержит ли какой-либо объект/класс указанное свойство. Функция называется «hasattr» и принимает два аргумента:

- проверяемый класс или объект;
- имя свойства, о существовании которого она сообщит (примечание: это должна быть строка с именем атрибута, а не просто имя)

Функция возвращает True или False.

Ее можно использовать следующим образом:

```
class ExampleClass:
    def __init__(self, val):
        if val % 2 != 0:
            self.a = 1
        else:
            self.b = 1


exampleObject = ExampleClass(1)
print(exampleObject.a)


if hasattr(exampleObject, 'b'):
    print(exampleObject.b)
```

Не забывайте, что функция hasattr() работает и с классами. Ее точно так же можно использовать, чтобы узнать, доступна ли переменная класса.

```
class ExampleClass:
    attr = 1

print(hasattr(ExampleClass, 'attr'))
print(hasattr(ExampleClass, 'prop'))
```

Функция возвращает True, если указанный класс содержит данный атрибут, в противном случае False.

Как вы думаете, что будет выведено на экран? Запустите код и проверьте, оказалось ли ваше предположение верным.

И еще один пример — посмотрите на код ниже и попытайтесь предсказать результат его вывода:

```
class ExampleClass:
    a = 1
    def __init__(self):
        self.b = 2

exampleObject = ExampleClass()

print(hasattr(exampleObject, 'b'))
print(hasattr(exampleObject, 'a'))
print(hasattr(ExampleClass, 'b'))
print(hasattr(ExampleClass, 'a'))
```

Получилось? Запустите код, чтобы проверить свои прогнозы.

Хорошо, мы добрались до конца этого раздела. В следующей теме мы с вами поговорим о методах, поскольку методы управляют объектами и заставляют их действовать.

Подробнее о методах

Давайте обобщим все, что касается использования методов в классах Python.

Как вы уже знаете, метод — это функция, встроенная в класс.

Существует одно фундаментальное требование — метод должен иметь хотя бы один параметр (методов без параметров не существует — метод можно вызвать без аргумента, но нельзя объявить без параметра).

Первый (или единственный) параметр обычно называется «self". Мы рекомендуем соблюдать это общепринятое правило, иначе, если вы решите использовать другие имена, вас ждет парочка неприятных сюрпризов.

Само имя self указывает на то, что делает этот параметр — он определяет объект, для которого вызывается метод,

Если нужно вызвать метод, нет необходимости передавать аргумент параметру self — Python сделает это за вас.

В примере ниже показана разница.

```
class Classy:
    def method(self):
        print("method")

obj = Classy()
obj.method()
```

Результат вывода:

```
method
```

Обратите внимание на то, как мы создали объект — мы использовали имя класса как функцию, вернув созданный объект класса.

Если вы хотите, чтобы метод принимал параметры, отличные от self, то следует:

- разместить их после self в определении метода;
- сгенерировать их во время вызова без указания параметра self (как выше).

Как здесь:

```
class Classy:
    def method(self, par):
        print("method:", par)

obj = Classy()
obj.method(1)
obj.method(2)
obj.method(3)
```

Результат вывода:

```
method: 1
method: 2
method: 3
```

Параметр self используется, чтобы получить доступ к экземпляру объекта и переменным класса,

В примере показаны оба способа использования параметра self:

```
class Classy:
    varia = 2
```

```
    def method(self):
        print(self.varia, self.var)
obj = Classy()
obj.var = 3
obj.method()
```

Результат вывода:

```
2 3
```

Параметр self используется и для вызова других методов объекта/класса внутри класса,

Как здесь:

```
class Classy:
    def other(self):
        print("other")
    def method(self):
        print("method")
        self.other()
obj = Classy()
obj.method()
```

Результат вывода:

```
method
other
```

Если назвать метод, к примеру, так: __init__, то это уже не обычный метод, а конструктор.

Если у класса есть конструктор, он вызывается автоматически и неявно, когда создается экземпляр объекта класса.

Конструктор:

- обязан иметь параметр self (задается автоматически, как обычно);
- может (но не обязан) иметь больше параметров (не только self); в этом случае в определении __init__ должно быть указано, каким образом имя класса используется для создания объекта;
- можно использовать для настройки объекта, т.е. правильно инициализировать его внутреннее состояние, создать переменные экземпляра, создать экземпляры любых других объектов при необходимости и т.д.

Посмотрите на код ниже.

```
class Classy:
    def __init__(self, value):
        self.var = value

obj1 = Classy("object")
print(obj1.var)
```

В примере показан очень простой конструктор. Запустите его. Результат вывода:

```
object
```

Обратите внимание, что конструктор:

- не может вернуть значение, так как он всего лишь должен вернуть созданный объект и не более;
- не может вызываться напрямую ни из объекта, ни из класса (конструктор можно вызвать из любого суперкласса объекта, но мы обсудим это позже).

Из-за того, что __init__ — это метод, а метод — это функция, с конструкторами/методами работают те же приемы, что и с обычными функциями.

В примере показано, как задать конструктор со значением аргумента по умолчанию.

```
class Classy:
    def __init__(self, value = None):
        self.var = value

obj1 = Classy("object")
obj2 = Classy()
print(obj1.var)
print(obj2.var)
```

Проверьте его.

Результат вывода:

```
object
None
```

Все, что мы говорили об именах свойств, относится и к именам методов — метод, имя которого начинается с __, (частично) скрыт.

В примере хорошо показан этот эффект:

```
class Classy:
    def visible(self):
        print("visible")
    def __hidden(self):
        print("hidden")

obj = Classy()
obj.visible()
```

```
try:
    obj.__hidden()
except:
    print("failed")

obj._Classy__hidden()
```

Результат вывода:

```
visible
failed
hidden
```

Запустите программу и протестируйте ее.

Внутренняя жизнь классов и объектов

У любого класса и объекта в Python есть набор полезных атрибутов, благодаря которым можно узнать, что умеет этот класс или объект.

Вы уже знаете один из таких атрибутов — это свойство __dict__.

Давайте посмотрим, как это работает с методами. Взгляните на код.

```
class Classy:
    varia = 1
    def __init__(self):
        self.var = 2
    def method(self):
        pass
    def __hidden(self):
        pass
```

```
obj = Classy()

print(obj.__dict__)
print(Classy.__dict__)
```

Запустите его и посмотрите на результат вывода. Внимательно проверьте вывод.

Найдите в коде все методы и атрибуты, которые содержат определение. Посмотрите, где именно находится это определение: внутри объекта или внутри класса.

__dict__ — это словарь. Стоит упомянуть еще одно встроенное свойство — __name__, которое является строкой.

Это свойство содержит название класса. Ничего необычного, это просто строка.

Обратите внимание, что атрибута __name__ нет в объекте, он существует только внутри классов.

Если надо найти класс конкретного объекта, можно использовать функцию с именем type(), которая (среди прочего) также может найти класс, из которого был создан экземпляр объекта.

Посмотрите на код, запустите его и убедитесь в этом сами.

```
class Classy:
    pass

print(Classy.__name__)
obj = Classy()

print(type(obj).__name__)
```

Результат вывода:

```
Classy Classy
```

Примечание: *такая строка кода: print(obj.__name__) вызовет ошибку.*
__module__ — тоже строка. Она хранит имя модуля, который содержит определение класса.

Давайте проверим, так ли это. Запустите код.

```
class Classy:
    pass
print(Classy.__module__)
obj = Classy()
print(obj.__module__)
```

Результат вывода:

```
__main__
__main__
```

Как известно, любой модуль с именем __main__ на самом деле не модуль, а файл, запущенный в данный момент.

__bases__ — это кортеж. Кортеж содержит классы (а не имена классов), которые являются прямыми суперклассами для класса.

Порядок такой же, как и в определении класса.

Мы покажем вам очень простой пример, поскольку хотим сделать акцент на том, как работает наследование.

Более того, мы покажем вам, как использовать этот атрибут, когда будем обсуждать объективные аспекты исключений.

Примечание: *только у классов есть этот атрибут, у объектов его нет.*

Мы определили функцию с именем printbases(), предназначенную для четкого представления содержимого кортежа.

Посмотрите на код ниже.

```
class SuperOne:
    pass

class SuperTwo:
    pass

class Sub(SuperOne, SuperTwo):
    pass

def printBases(cls):
    print('( ', end='')

    for x in cls.__bases__:
        print(x.__name__, end=' ')
    print(')')

printBases(SuperOne)
printBases(SuperTwo)
printBases(Sub)
```

Проанализируйте и запустите его. Результат вывода следующий:

```
( object )
( object )
( SuperOne SuperTwo )
```

Примечание: *класс без явных суперклассов указывает на объект (заранее определенный класс Python) как на своего прямого предка.*

Рефлексия и интроспекция

Все эти средства позволяют Python-программисту выполнять два важных действия, которые характерны для многих объектных языков. А именно:

- **интроспекция** — способность программы проверять тип или свойства объекта во время выполнения;
- **рефлексия (отражение)** — способность программы манипулировать значениями, свойствами и/или функциями объекта во время выполнения.

Другими словами, вам не нужно знать полное определение класса/объекта, чтобы манипулировать объектом, поскольку объект и/или его класс содержат метаданные, позволяющие распознавать его функции во время выполнения программы.

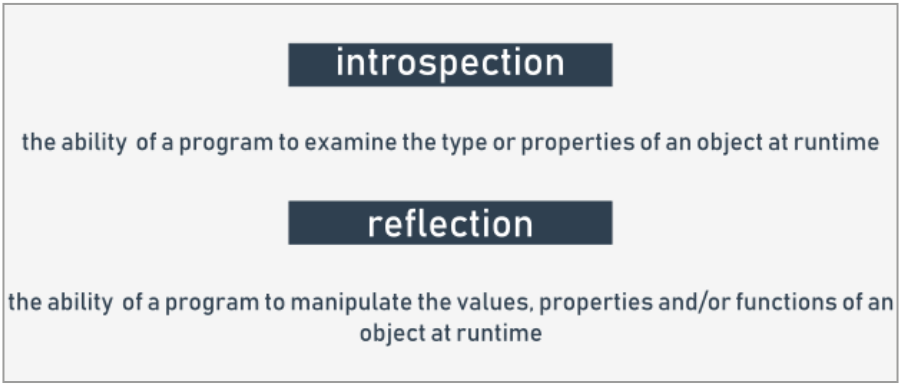

Рисунок 9

Анализ классов

Что Python позволяет узнать о классах? Ответ прост — все.

Благодаря рефлексии и интроспекции программист может делать что угодно с любым объектом.

Проанализируйте следующий код.

```
class MyClass:
    pass

obj = MyClass()
obj.a = 1
obj.b = 2
obj.i = 3
obj.ireal = 3.5
obj.integer = 4
obj.z = 5

def incIntsI(obj):
    for name in obj.__dict__.keys():
        if name.startswith('i'):
            val = getattr(obj, name)
            if isinstance(val, int):
                setattr(obj, name, val + 1)

print(obj.__dict__)
incIntsI(obj)
print(obj.__dict__)
```

Функция с именем incIntsI() получает объект любого класса, просматривает его содержимое, чтобы найти все целочисленные атрибуты с именами, которые начинаются на i, и увеличивает их значение на единицу.

Думаете, это невозможно? Напротив!

Вот как это работает:

- *строка 1:* определить очень простой класс...
- *строки с 3 по 10:* ... и заполнить его атрибутами;
- *строка 12:* это наша функция!
- *строка 13:* сканировать атрибут __dict__ на предмет всех имен атрибута;
- *строка 14:* если имя начинается с i...
- *строка 15:* ... использовать функцию getattr(), чтобы получить ее текущее значение; примечание: getattr() принимает два аргумента: объект и имя его свойства (в виде строки) и возвращает значение текущего атрибута;
- *строка 16:* проверить, является ли значение integer, и использовать для этого функцию isinstance() (мы обсудим это позже);
- *строка 17:* если проверка прошла успешно, увеличить значение свойства, используя функцию setattr(); функция принимает три аргумента: объект, имя свойства (в виде строки) и новое значение свойства.

Результат вывода:

```
{'a': 1, 'integer': 4, 'b': 2, 'i': 3, 'z': 5,
 'ireal': 3.5}
{'a': 1, 'integer': 5, 'b': 2, 'i': 4, 'z': 5,
 'ireal': 3.5}
```

Вот и все!