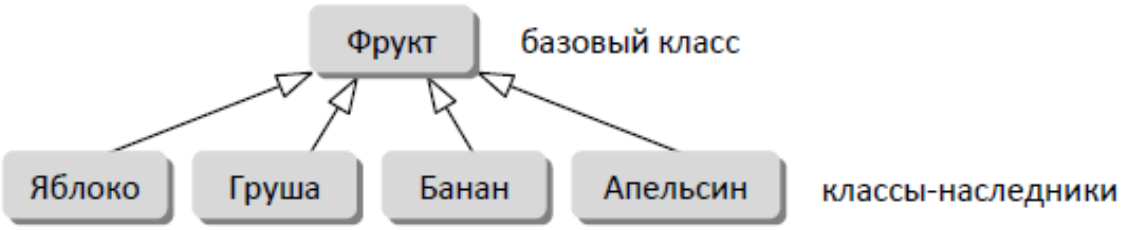
Иерархия классов

Классификации

Как в науке, так и в быту, важную роль играет *классификация* – разделение изучаемых объектов на группы (классы), объединенные общими признаками. Прежде всего, это нужно для того, чтобы не запутаться в большом количестве данных и не описывать каждый объект заново.

Например, есть много видов фруктов (яблоки, груши, бананы, апельсины и т.д.), но все они обладают некоторыми общими свойствами. Если перевести этот пример на язык ООП, класс *Яблоко* – это подкласс (производный класс, класс-наследник, потомок) класса Фрукт, а класс Фрукт –это базовый класс (суперкласс, класс-предок) для класса *Яблоко* (а также для классов *Груша*, *Банан*, *Апельсин* и других).



Стрелка с белым наконечником на схеме обозначает наследование. Например, класс *Яблоко* – это наследник класса *Фрукт*.

Классический пример научной классификации – классификация животных или растений. Как вы знаете, она представляет собой *иерархию* (многоуровневую структуру). Например, горный клевер относится к роду *Клевер* семейства Бобовые класса *Двудольные* и т.д. Говоря на языке ООП, класс *Горный клевер* – это наследник класса Клевер, а тот, в свою очередь, наследник класса *Бобовые*, который также является наследником класса *Двудольные* и т.д.

Класс Б является наследником класса А, если можно сказать, что Б – это разновидность А.

Например, можно сказать, что яблоко – это фрукт, а горный клевер – одно из растений семейства *Двудольные*.

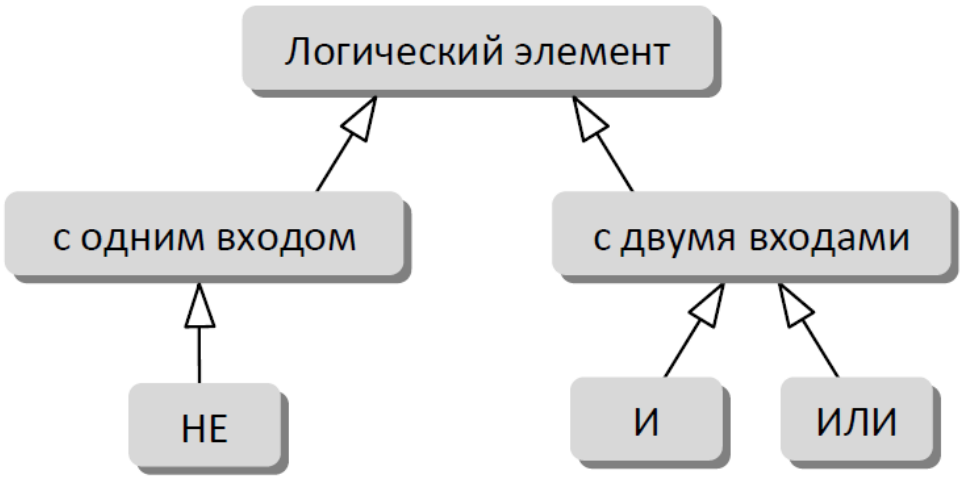
В то же время мы не можем сказать, что «машина – это разновидность двигателя», поэтому класс *Машина* не является наследником класса *Двигатель*. *Двигатель* – это составная часть машины, поэтому объект класса *Машина* содержит в себе объект класса *Двигатель*. Отношения между двигателем и машиной – это отношение «часть – целое»

Иерархия логических элементов

Рассмотрим такую задачу: составить программу для моделирования управляющих схем, построенных на логических элементах. Нам нужно «собрать» заданную схему и построить ее таблицу истинности.

Как вы уже знаете, перед тем, как программировать, нужно выполнить объектно-ориентированный анализ. Все объекты, из которых состоит схема – это логические элементы, однако они могут быть разными («НЕ», «И», «ИЛИ» и другие). Попробуем выделить общие свойства и методы всех логических элементов.

Ограничимся только элементами, у которых один или два входа. Тогда иерархия классов может выглядеть так:



Среди всех элементов с двумя входами мы показали только элементы «И» и «ИЛИ», остальные вы можете добавить самостоятельно.

Итак, для того, чтобы не описывать несколько раз одно и то же, классы в программе должны быть построены в виде иерархии. Теперь можно дать классическое определение объектно-ориентированного программирования:

Объектно-ориентированное программирование – это такой подход к программированию, при котором программа представляет собой множество взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

Базовый класс

Построим первый вариант описания класса *Логический элемент* (**`TLogElement`**). Обозначим его входы как **`In1`** и **`In2`**, а выход назовем **`Res`** (от англ. *result* – результат). Здесь состояние логического элемента определяется тремя величинами (**`In1`**, **`In2`** и **`Res`**). С помощью такого базового класса можно моделировать не только статические элементы (как «НЕ», «И», «ИЛИ» и т.п.), но и элементы с памятью (например, триггеры). Для сохранения значений входов и запоминания выхода введём три поля, принимающих логические значения и заполним их в конструкторе:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
```

Названия полей **__in1** и **__in2** начинаются с двух подчеркиваний, поэтому они будут скрытыми (доступны только внутри методов класса **TLogElement**). Имя поля **_res** начинается с одного подчёркивания, то есть оно не скрывается. В то же время одно подчёркивание говорит о его специальной роли, мы вернёмся к этому чуть позже.

Для доступа к полям введём свойства **In1**, **In2** и **Res** (свойство только для чтения):

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
  def __setIn1( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
  def __setIn2( self, newIn2 ):
    self.__in2 = newIn2
    self.calc()
  In1 = property( lambda x: x.__in1, __setIn1 )
  In2 = property( lambda x: x.__in2, __setIn2 )
  Res = property( lambda x: x._res )
```

Методы чтения для всех свойств записаны как «лямбда-функции», они выполняют прямой.

Вы, наверное, заметили, что оба метода записи после присваивания нового значения входу вызывают какой-то метод **calc**, которого нет в описании класса. В то же время видно, что этот метод принадлежит классу, потому что перед его именем добавлена ссылка **self**.

Метод **calc** должен пересчитать значение выхода логического элемента сразу после изменения его входа. Проблема в том, что мы не можем написать метод **calc**, пока неизвестно, какой именно логический элемент моделируется. С другой стороны, мы знаем, что такую процедуру имеет любой логический элемент. В такой ситуации можно написать метод-«заглушку» (который ничего не делает):

```
class TLogElement:
  ...
  def calc( self ):
    pass
```

Но это не совсем правильно, поскольку кто-то может создать такой элемент и пытаться его использовать, а он не работает! Поэтому не будем его определять вообще. Такой метод называется *абстрактным методом*.

Абстрактный метод – это метод класса, который используется, но не реализуется в классе.

Более того, не существует логического элемента «вообще», как не существует «просто фрукта», не относящегося к какому-то виду. Такой класс в ООП называется

абстрактным. Его отличительная черта – хотя бы один абстрактный (нереализованный) метод.

Абстрактный класс – это класс, содержащий хотя бы один абстрактный метод.

Для надёжности нужно совсем запретить создание объектов класса **TLogElement**, потому что это бессмысленно. Для этого в конструкторе класса определим, есть ли у объекта метод **calc**, и если нет – будем искусственно воздавать аварийную ситуацию – исключение:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
    if not hasattr( self, "calc" ):
      raise NotImplementedError("Нельзя создать такой объект!")
```

Функция **hasattr** возвращает логическое значение **True**, если у переданного ему объекта (здесь – **self**) есть указанное поле или метод (**calc**). Ключевое слово **raise** означает «создать исключение», тип этого исключения – **NotImplementedError** (ошибка «не реализовано»), в скобках записана символьная строка, которую увидит пользователь в сообщении об ошибке.

Итак, полученный класс **TLogElement** – это абстрактный класс. Его можно использовать только для разработки классов-наследников, создать в программе объект этого класса нельзя. Чтобы класс-наследник не был абстрактным, он должен переопределить все абстрактные методы предка, в данном случае – метод **calc**. Как это сделать, вы увидите в следующем пункте.

Получается, что классы-наследники могут по-разному реализовать один и тот же метод. Такая возможность называется *полиморфизм*.

Полиморфизм (от греч. πολυ — много, и μορφη — форма) – это возможность классов-наследников по-разному реализовать метод, описанный для класса-предка.

Теперь давайте вспомним про поле с именем **_res**, которое хранит значение выхода логического элемента. Очевидно, что оно должно быть доступно классам-наследникам, которые будут изменять его в методе **calc**. Поэтому делать его скрытым нельзя. С другой стороны, часто используют соглашение о том, что одно подчёркивание означает специальное поле, которое недолжно изменяться другими объектами и функциями.

Классы-наследники

Теперь займемся классами-наследниками от **TLogElement**. Поскольку у нас будет единственный элемент с одним входом («НЕ»), сделаем его наследником прямо от **TLogElement** (не будем вводить специальный класс «элемент с одним входом»).

```
class TNot ( TLogElement ):
  def __init__ ( self ):
    TLogElement.__init__ ( self )
  def calc ( self ):
    self._res = not self.In1
```

После названия нового класса **TNot** в скобках указано название базового класса. Все объекты класса **TNot** обладают всеми свойствами и методами класса **TLogElement**.

В конструкторе класса-наследника нужно обязательно вручную вызвать конструктор класса-предка. В отличие от других объектно-ориентированных языков, этот вызов автоматически не выполняется.

Новый класс определяет метод **calc**, который записывает в поле **_res** инверсию входного сигнала (результат применения логической операции **not**). Таким образом, класс **TNot** уже неабстрактный, в нём все нужные методы определены. Теперь можно создавать объект этого класса **TNot** и использовать его:

```
n = TNot()
n.In1 = False
print ( n.Res )
```

Все типы логических элементов, которые имеют два входа, будут наследниками класса

```
class TLog2In ( TLogElement ):
  pass
```

В нашем случае этот класс пустой, но если нам понадобится ввести какие-то свойства и методы, характерные для всех элементов с двумя входами, мы сможем это легко сделать в классе **TLog2In**.

Класс **TLog2In** – это тоже абстрактный класс, потому что он не переопределил метод **calc**. Это сделают его наследники **TAnd** (элемент «И») и **TOr** (элемент «ИЛИ»), которые описывают конкретные логические элементы:

```
class TAnd ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 and self.In2
class TOr ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 or self.In2
```

Теперь мы готовы к тому, чтобы создавать и использовать построенные логические элементы. Например, таблицу истинности для последовательного соединения элементов «И» и «НЕ» можно построить так:

```
elNot = TNot()
elAnd = TAnd()
print( "  A | B | not(A&B) " );
print( "-------------------" );
for A in range(2):
  elAnd.In1 = bool(A)
  for B in range(2):
    elAnd.In2 = bool(B)
    elNot.In1 = elAnd.Res
    print( " ", A, "|", B, "|", int(elNot.Res) )
```

Сначала создаются два объекта – логические элементы «НЕ» (класс **`TNot`**) и «И» (класс **`TAnd`**). Далее в двойном цикле перебираются все возможные комбинации значений переменных A и B (каждая из них может быть равна 0 или 1). Они подаются на входы элемента «И», а его выход – на вход элемента «НЕ». Чтобы при выводе таблицы истинности вместо **`False`** и **`True`** выводились более компактные обозначения 0 и 1, значение выхода преобразуется к целому типу (**`int`**).

Модульность

Большие программы обычно разбивают на модули – внутренне связные, но слабо связанные между собой блоки. Такой подход используется как в классическом программировании, так в ООП.

В нашей программе с логическими элементами в отдельный модуль (сохраним его в виде файла **`logelement.py`**) можно вынести всё, что относится к логическим элементам:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
    if not hasattr( self, "calc" ):
      raise NotImplementedError("Нельзя создать такой объект!")
  def __setIn1( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
  def __setIn2( self, newIn2 ):
    self.__in2 = newIn2
    self.calc()
  In1 = property( lambda x: x.__in1, __setIn1 )
  In2 = property( lambda x: x.__in2, __setIn2 )
  Res = property( lambda x: x._res )
```

```
class TNot ( TLogElement ):
  def __init__ ( self ):
    TLogElement.__init__ ( self )
  def calc ( self ):
    self._res = not self.In1
class TLog2In ( TLogElement ):
  pass
class TAnd ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 and self.In2
class TOr ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 or self.In2
```

Чтобы использовать такой модуль, нужно подключить его в основной программе с помощью ключевого слова **`import`**:

```
import logelement
elNot = logelement.TNot()
elAnd = logelement.TAnd()
...
```

Обратите внимание, что при создании объектов нужно указывать имя модуля, где определены классы **`TNot`** и **`TAnd`**.

Сообщения между объектами

Когда логические элементы объединяются в сложную схему, желательно, чтобы передача сигналов между ними при изменении входных данных происходила автоматически. Для этого можно немного расширить базовый класс **`TLogElement`**, чтобы элементы могли передавать друг другу сообщения об изменении своего выхода.

Для простоты будем считать, что выход любого логического элемента может быть подключен к любому (но только одному!) входу другого логического элемента. Добавим к описанию класса два скрытых поля и один метод:

```
class TLogElement:
  def __init__ ( self ):
    ...
    self.__nextEl = None
    self.__nextIn = 0
  ...
  def link ( self, nextEl, nextIn ):
    self.__nextEl = nextEl
    self.__nextIn = nextIn
```

Поле **`__nextEl`** хранит ссылку на следующий логический элемент, а поле **`__nextIn`** – номер входа этого следующего элемента, к которому подключен выход данного

элемента. С помощью общедоступного метода **link** можно связать данный элемент со следующим.

Нужно немного изменить методы setIn1 и setIn2: при изменении входа они должны не только пересчитывать выход данного элемента, но и отправлять сигнал на вход следующего

```
class TLogElement:
  ...
  def __setIn1( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
```

```
    if self.__nextEl:
      if self.__nextIn == 1:
        self.__nextEl.In1 = self._res
      elif __nextIn == 2:
        __nextEl.In2 = self._res
```

Запись «**if __nextEl**» означает «если следующий элемент задан». Если он не был установлен, значение поля **__nextEl** будет равно **None**, и никаких дополнительных действий не выполняется.

С учетом этих изменений вывод таблицы истинности функции «И-НЕ» можно записать так (операторы вывода заменены многоточиями):

```
elNot = logelement.TNot()
elAnd = logelement.TAnd()
elAnd.link( elNot, 1 )
...
for A in range(2):
  elAnd.In1 = bool(A)
  for B in range(2):
    elAnd.In2 = bool(B)
    ...
```

Обратите внимание, что в самом начале мы установили связь элементов «И» и «НЕ» с помощью метода **link** (связали выход элемента «И» с первым входом элемента «НЕ»). Далее в теле цикла обращения к элементу «НЕ» нет, потому что элемент «И» автоматически сообщит ему об изменении своего выхода.