**Дефинирование и создание простых классов**

Если программисту надо передать много взаимосвязанных частей данных функции, может быть сложным использовать только переменные. Например, каждый раз, когда нам нужно передать адрес, было бы утомительным передавать имя, улицу, город, страну и почтовый код. Лучше задать структуру данных, которая будет содержать всю информацию вместе, а затем присваивать данным имена, такие как *Address*. Этого можно легко достигуть в Python или любом другом современном языке используя классы, *class*.

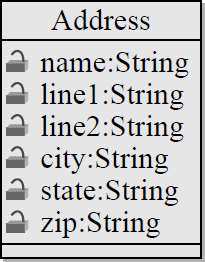
Это пример кода класса, содержащего адрес:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Address():  name=""  line1=""  line2=""  city=""  state=""  zip="" |

В коде, приведённом выше, **Address** является *именем класса*. Переменные в классе, такие, как **name** и**city**, называются *атрибутами* или *полями*.

Заметьте сходства с созданием дефиниции функции. Код *задаёт* класс, но он не создаёт *инстанцию*класса. Код сказал компьютеру, какие поля есть у адреса, и какие у них будут начальные значения.

В отличие от функции, имена классов должны начинаться с большой буквы. Хотя можно начинать имена классов и с маленькой буквы, это не считается хорошим стилем.Программисты часто рисуют диаграммы классов, а диаграмма этого класса будет смотреться следующим образом:



Имя класса вверху, атрибуты находятся под ним. Справа от атрибута написан тип данных, такой как string или integer.

Заметьте, что, хотя код и дефинирует адрес, никакого адреса создано не было. Также, как и с функцией, дефинирование функции не означает её вызов. Чтобы создать класс и установить поля, ознакомьтесь с нижепривежённым примером:

|  |  |
| --- | --- |
| 9  10  11  12  13  14  15  16  17 | # Создать инстанцию класса address  homeAddress=Address()  # Задать поля в address  homeAddress.name="John Smith"  homeAddress.line1="701 N. C Street"  homeAddress.line2="Carver Science Building"  homeAddress.city="Indianola"  homeAddress.state="IA"  homeAddress.zip="50125" |

Инстанция класса address была создана на 9й строке. Заметьте, что имя класса используется вместе со скобками. Именем переменной может быть любым до тех пор, пока оно придерживается нормальных правил наименования переменных.

Для задачи полей в классе, программа использует оператор-точку. Этот оператор - точка, между**homeAddress** и самим полем. Левое поле присваивание кажется нормальным. Линии с 12 по 17 задают значения полей.

Обычная ошибка при работе с классами: забывать задавать переменную. Если был создан один адресс, обычное дело предположить, что компьютер будет знать, что нужно использовать этот адрес при задачи города “Indianola”. Однако, это так не работает.

Второй адрес может быть создан, теперь могут быть использованы поля обеих классов. Продолжая предыдущий пример, ознакомьтесь с нижеприведённым кодом:

|  |  |
| --- | --- |
| 18  19  20  21  22  23  24  25 | # Создать другой адресс  vacationHomeAddress=Address()  # Задать поля этого адреса  vacationHomeAddress.name="John Smith"  vacationHomeAddress.line1="1122 Main Street"  vacationHomeAddress.line2=""  vacationHomeAddress.city="Panama City Beach"  vacationHomeAddress.state="FL"  vacationHomeAddress.zip="32407"  print ("The client's main home is in "+homeAddress.city)  print ("His vacation home is in "+vacationHomeAddress.city) |

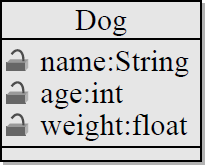
Строка 19 создаёт второй адрес и присваивает его новой переменной. Строки 22-27 дают значения полям инстанции нового класса. Строка 29 выводит город домашнего адреса потому, что**homeAddress** появляется после оператора-точки. Строка 30 выводит отпускной адрес потому, что**vacationHomeAddress** появляется перед оператором-точкой.

В вышеприведённом примере, **Address** является классом *class*, потому что он задаёт новую классификацию объекту данных. Переменные **homeAddress** и **vacationHomeAddress** обращаются к*объектам*, потому что они ссылаются на настоящие инстанции класса **Address**. Объект - это просто интснция класса.

Перемещае множество полей с данными в класс делает передачу данных в функцию и из неё простым делом. В нижеприведённом коде, функция берёт адрес как параметр и печатает его на экран. Не обязательно передавать параметр для каждого поля адреса.

|  |  |
| --- | --- |
| 31  32  33  34  35  36  37  38  39  40  41 | # Вывести адрес на экран  def printAddress(address):  print (address.name)  # If there is a line1 in the address, print it  if( len(address.line1) > 0 ):  print (address.line1)  # If there is a line2 in the address, print it  if( len(address.line2) > 0 ):  print( address.line2 )  print( address.city+", "+address.state+" "+address.zip )  printAddress( homeAddress )  print()  printAddress( vacationHomeAddress ) |

1.1 Проверка пройденного

1. Напишите код для создания инстанции следующего класса и задайте его атрибуты:   
   

|  |
| --- |
| class Dog():  age=0  name=""  weight=0    **class** Dog():  age=0  name=**" "**  weight=o  Rex = Dog()  Rex.age = 5  Rex.name = Rexik  Rex.weight = 30 |

1. Создайте код для созданиях *двух разных* инстанций этого класса и задайте атрибуты обоих объектов:

|  |
| --- |
| class Person():  name=""  cellPhone=""  email=""    **class** Person():  name=**''**  cellPhone=**''**  email=**''**  Elena=Person()  Elena.name=**'Elena Petrovna'**  Elena.cellPhone=**'12345678'**  Elena.email=**'qwerty@mail.ru'**  Igor=Person()  Igor.name=**'Igor Grigorevich'**  Igor.cellPhone=**'87654321'**  [Igor.email=**'ytrewq@mail.ru**](mailto:Igor.email%3D'ytrewq@mail.ru)**'** |

1. Для нижеприведённого кода, напишите класс, который содержит подходящее имя класса и атрибуты, которые позволят коду работать.

|  |
| --- |
| myBird = Bird()  myBird.color="green"  myBird.name="Sunny"  myBird.breed="Sun Conure"    **class** Bird():  color=**''**  name=**''**  breed=**''** |

1. Создайте класс, который будет представлять из себя персонажа простой 2D игры. Включите атрибуты для расположения на экране, имени, силы.

**class** Personazh():

name=**''**

gor\_X=0

ver\_Y=0

sila=0

1. Следующий код запускается, но неправильно работает. Что программист сделал неправильно?

|  |
| --- |
| class Person:  name=""  money=0  nancy=Person()  name="Nancy"  money=100  Программист использовал не абъекты класса, а переменные  **class** Person():  name=**""**  money=0  nancy=Person()  nancy.name=**"Nancy"**  nancy.money=100 |

1. Ознакомьтесь с кодом. Он не запускается. Можете ли вы заметить ошибку?

|  |
| --- |
| class Person:  name=""  money=0  bob = Person()  print (bob.name,"has",money,"dollars.")    **class** Person:  name=**""**  money=0  bob = Person()  **print** (bob.name,**"has"**,bob.money,**"dollars."**) |

1. Даже с исправленной ошибкой, программа не покажет желаемый вывод:   
   **Bob has 0 dollars.**

Почему?

Потому что атрибуты не имели конкретных значений.

**class** Person:

name=**''**

money=0

bob = Person()

bob.name=**'Bob'**

bob.money=0

**print** (bob.name,**'has'**,bob.money,**'dollars.'**)

1.2 Методы

Помимо атрибутов, у класса могут быть *методы*. Метод - это функция, которая существует внутри класса. Расширяя предыдущий пример класса Dog, нижеприведённый код добавляет метод лая.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | class Dog():  age=0  name=""  weight=0  def bark(self):  print( "Woof" ) |

Дефиниция метода показана на строках 6-7. Дефиниции метода в классе смотрятся так же, как и дефиниции функций. Большое отличие в том, что добавляется параметр **self** на строке 6. Первым параметром любого метода в классе должен быть **self**. Этот параметр требуется, даже если функция его не использует.

Вот важные пункты, которые следует держать в голове, создавая методы для классов:

* Сначала следует перечислить атрибуты, а лишь потом - методы.
* Первым параметром любого метода должен быть **self**.
* Дефиниции методов сдвинуты ровно на один Tab.

Методы должны быть вызваны в схожем с присвоением атрибутов объекту стилем. Ознакомьтесь с нижеприведённым кодом.

|  |  |
| --- | --- |
| 9  10  11  12  13  14  15 | myDog = Dog()  myDog.name="Spot"  myDog.weight=20  myDog.age=3  myDog.bark() |

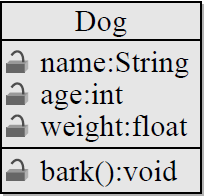
На 9й строке "создаётся" собака. Строки 11-13 задают атрибуты объекта. Строка 14 вызывает метод**bark**. Заметьте: несмотря на то, что у функции **bark** есть один параметр, **self**, вызов не передаёт ей ничего. Это потому, что первый параметр является ссылкой на сам объект dog. На самом деле, при обработке кода Python делает вызов вроде

|  |
| --- |
| # Просто пример, не работает  Dog.bark(myDog) |

Если функции **bark** требуется сослаться на один из атрибутов, она делает это используя переменную **self**. Например, мы можем поменять класс **Dog** так, чтобы когда собака лает, он также будет выводить её имя. В нижеприведённом коде, доступ к атрибуту имени осуществляется через переменную **self**.

|  |  |
| --- | --- |
| 6  7 | def bark(self):  print( "Woof says",self.name ) |

Атрибуты - прилагательные, методы - глаголы. Диаграмма класса выглядела бы следующим образом:



13.1.3 Пример: класс Ball

Это пример кода, который можно использовать в Python/Pygame для рисования мячика. Содержание всех параметров в классе делает управление данными проще.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class Ball():  # --- Атрибуты класса ---  # Ball position  x=0  y=0  # Ball's vector  change\_x=0  change\_y=0  # Ball size  size=10  # Ball color  color=[255,255,255]  # --- Методы класса ---  def move(self):  self.x += self.change\_x  self.y += self.change\_y  def draw(self, screen):  pygame.draw.circle(screen, self.color, [self.x, self.y], self.size ) |

Ниже - код, который выполнится перед главным циклом программы для создания шара и задания его атрибутов:

|  |
| --- |
| theBall = Ball()  theBall.x = 100  theBall.y = 100  theBall.change\_x = 2  theBall.change\_y = 1  theBall.color = [255,0,0] |

Этот код пойдёт внутрь главного цикла чтобы двигать и рисовать шар:

|  |
| --- |
| theBall.move()  theBall.draw(screen) |

2 Ссылки

Ознакомьтесь с нижеприведённым кодом:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | class Person:  name=""  money=0  bob = Person()  bob.name="Bob"  bob.money=100 |

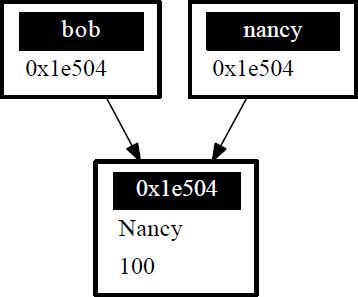
Обычным заблуждением при работе с объектами является предположение, что переменная **bob***является* объектом Person. Это не так. Переменная **bob** - *ссылка* на объект Person. Это показано дополняя наш пример кода и запуская его:

|  |  |
| --- | --- |
| 9  10  11  12  13 | nancy = bob  nancy.name="Nancy"  print(bob.name,"has",bob.money,"dollars.")  print(nancy.name,"has",nancy.money,"dollars.") |

Если **bob** действительно был бы объектом, то строка 9 могла бы создать *копию* объекта и у нас уже было бы два разных объекта. Вывод программы показал бы, что у Боба и Нэнси есть 100 долларов. Но, при запуске, программа выводит следующую строку:

**Nancy has 100 dollars.**

**Nancy has 100 dollars.**



То, что хранит **bob** - это *ссылка* на объект. Другим названием этого может быть *адрес*, *указатель*или *handle*. Это адрес в компьютерной памяти, где хранится сам объект. Адрес хранится в шестнадцатиричном формате, который при выводе будет выглядеть как-то вроде 0x1e504. Когда выполняется девятая строка, копируется адрес, а не сам объект, на который указывает адрес.

2.1 Функции и ссылки

Нижеприведённый код схож с примером, показаным раньше. Код на первой строке создаёт функцию, берущую число. Переменная **money** - копия того, что передаётся. Прибавить к этому число 100 не изменит значения, хранимого в **bob.money** на строке 10. Поэтому print на 13й строке выводит 100, а не 200.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def giveMoney1(money):  money += 100  class Person:  name=""  money=0  bob = Person()  bob.name="Bob"  bob.money=100  giveMoney1(bob.money)  print(bob.money) |

Посмотрите на дополнительный код снизу. Этот код действительно заставляет переменную**bob.money** увеличиться, а затем выражение print выведет 200.

|  |  |
| --- | --- |
| 14  15  16  17  18 | def giveMoney2(person):  person.money += 100  giveMoney2(bob)  print(bob.money) |

Почему именно так? Потому, что **person** содержит адрес объекта в памяти, а не сам объект. Можно воспринимать это как номер своего банковского счёта. У функции есть копия номера банковского счёта, но не копия самого счёта. Так что после использования номера банковского счёта для того, чтобы разместить на нём 100 долларов, количество денег на счету Боба увеличивается.

Массивы работают по такому же принципу. Функция, которая берёт массив (list) как параметр и меняет его значения, будет менять значения того же самого массива, используемого в коде. Копируется адрес массива, но не сам массив.

2.2 Проверка пройденного

1. Создайте класс **Cat**. Дайте ему аттрибуты name(имя), color(цвет) и weight(вес). Добавьте ему метод под названием **meow**.
2. Создайте объект класса cat, установите атрибуты, вызовите метод meow.
3. Создайте класс **Monster**. Дайте ему атрибуты name(имя) и целочисленный атрибут health(здоровье). Создайте метод **decreaseHealth** который берёт числовой параметр **amount** и уменьшает здоровье на переданное количество. Внутри метода, сделайте вывод строки, оповещающей о смерти монстра, если его здоровье упало ниже нуля.

**class** Monster():

name=**''**

health=0

**def** decreaseHealth(self,amount):

self.health = self.health-amount

**if** (self.health<=0):

**print** (**'monstr mertv'**)

3 Конструкторы

Классы в Python имеют особую функцию, вызываемую в момент создания инстанции класса. Например, ниже приведён класс **Dog**, продолжающийся кодом, создающим инстанцию этого класса:

|  |
| --- |
| class Dog()  name=""  myDog = Dog() |

Добавив функцию под названием *constructor*, программист может добавить код, автоматически вызываемый каждый раз во время создании инстанции класса. Посмотрите пример конструктора ниже:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Dog():  name=""  # Конструктор  # Вызывается в момент создания объекта этого типа  def \_\_init\_\_(self):  print("A new dog is born!")  # This creates the dog  myDog = Dog() |

Конструктор начинается на 6й строке. Он должен быть назван **\_\_init\_\_**. Он должен брать **self** как первый параметр. Дополнительно можно задать ещё параметров. Когда программа запустится, она выведет:   
**A new dog is born!**   
Когда объект типа **Dog** создаётся на 10й строке, автоматически вызывается функция **\_\_init\_\_**, которая затем выводит сообщение на экран.

Конструктор может быть использован для инициализации и задании данных объекта. Например, в примере кода атрибут **name** оставлен пустым после создания объекта dog. Что, если программист захочет дать начальное значение одному из аттрибутов объекта? Некоторые объекты нуждаются в значениях атрибутов с самого начала, на момент их создания. Конструктор может быть использован для осуществления таких задач. Ознакомьтесь с кодом ниже:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Dog():  name=""  # Конструктор  # Вызывается на момент создания объекта этого типа  def \_\_init\_\_(self, newName):  self.name = newName  # Это создаёт собаку  myDog = Dog("Spot")  # Вывести имя собаки, убедиться, что оно было установлено  print(myDog.name)  # Эта строка выдаст ошибку, потому что  # конструктору не было передано имя  herDog = Dog() |

На шестой строке программе даются дополнительные параметры **newName**. Значение этого параметра используется для установления значения атрибута **name** в классе **Dog**. Частая ошибка заключается в установлении имён параметров функции **\_\_init\_\_** такими же как и имена атрибутов, предполагая, что атрибуты автоматически возьмут значение переданных параметров. Этого не случится.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | class Dog():  name="Rover"  # Конструктор  # Вызывается во время создания объекта  def \_\_init\_\_(self, name):  # This will print "Rover"  print(self.name)  # This will print "Spot"  print(name)  # Это создаёт собаку  myDog = Dog("Spot") |

В предыдущем примере, выводятся две переменных. Переменная **name** была создана как параметр метода на шестой строке. Эта переменная пропадёт как только метод закончит свою работу. Хотя у неё такое же имя, как и у атрибута **name**, (также называемого *объектной переменной*), она является абсолютно другой переменной. Переменная **self.name** ссылается на атрибут **name** этой конкретной инстанции класса **Dog**. Она будет существовать пока существует этот объект класса **Dog**.

3.1 Проверка пройденного

1. Должны ли имена класса начинаться с маленькой или большой буквы?

Имя класса пишется с большой буквы

1. Должны ли имена методов класса начинаться с маленькой или большой буквы?

Не имеет значения

1. Что в классе следует перечислить сначала - атрибуты или методы?

Сначала перечисляются атрибуты потом методы

1. Как иначе можно назвать ссылку?

Адрес в памяти

1. Какое другое название переменной инстанции?

Aтрибут

1. Как иначе можно назвать инстанцию класса?

Объект класса

1. Создайте класс под названием **Star**, который будет выводить “A star is born!” каждый раз во время его создания.

**class** Star():

**def** \_\_init\_\_(self):

**print**(**"A star is born!"**)

myStar = Star()

1. Создайте класс **Monster** с атрибутами для здоровья и имени. Добавьте конструктор к этому классу, устанавливающий здоровья и имя равными значениям, переданным как параметры.

**class** Monster():

name=**''**

health=0

**def** \_\_init\_\_(self, newName, newHealth):

self.name=newName

self.health=newHealth

4 Наследование

Другой мощной особенностью использования классов и объектов является возможность*наследования*. Можно создать класс и унаследовать все атрибуты и методы *родительского класса*. Например, программа может создавать класс **Person**, содержащий все атрибуты, нужные для описания человека. Затем, программа может создавать *дочерние классы*, наследующие эти поля. Дочерние классы могут добавлять свои поля и методы, соответствующие их нуждам.

Родительский класс должен быть более обобщённой, абстрактной версией дочернего класса. Например, родительский класс может быть Animal(животное), а дочерним классом Dog(собака), которая также может иметь свой, дочерний класс Poodle(пудель). Это отношение ребёнка-родителя объясняется с помощью *является*. Например дельфин *является* млекопитающим. Это не работает в обратном направлении, потому что млекопитающее не обязательно должно быть дельфином. Поэтому класс Dolphin никогда не должен быть родителем класса Mammal. Подобно этому, класс Table не должен быть родителем класса Chair, потому что стул не является столом.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | class Person():  name=""  class Employee(Person):  job\_title=""  class Customer(Person):  email=""  johnSmith = Person()  johnSmith.name = "John Smith"  janeEmployee = Employee()  janeEmployee.name = "Jane Employee"  janeEmployee.job\_title = "Web Developer"  bobCustomer = Customer()  bobCustomer.name = "Bob Customer"  bobCustomer.email = "send\_me@spam.com" |

Поставив **Person** между скобок на строках 4 и 7, программист сказал компьютеру, что **Person**является родительским классом как для **Employee**(работника), так и для **Customer**(клиента). Это позволяет программе задать атрибут **name** на строках 14 и 18.

Методы также наследуются. Нижеприведённый код выведет “Person created” три раза, потому что классы клиента и работника наследуют конструктор от родительского класса:

|  |
| --- |
| class Person():  name=""  def \_\_init\_\_(self):  print("Person created")  class Employee(Person):  job\_title=""  class Customer(Person):  email=""  johnSmith = Person()  janeEmployee = Employee()  bobCustomer = Customer() |

Методы могут быть *перекрытыми* дочерним классом, предоставляя им новый функционал. В нижеприведённом коде, дочерние классы содержат свои конструкторы, поэтому родительский конструктор запущен не будет:

|  |
| --- |
| class Person():  name=""  def \_\_init\_\_(self):  print("Person created")  class Employee(Person):  job\_title=""  def \_\_init\_\_(self):  print("Employee created")  class Customer(Person):  email=""  def \_\_init\_\_(self):  print("Customer created")  johnSmith = Person()  janeEmployee = Employee()  bobCustomer = Customer() |

Если программист захочет вызвать метод как родительского, так и дочернего класса, дочерний класс может вызвать родительский метод:

|  |
| --- |
| class Person():  name=""  def \_\_init\_\_(self):  print("Person created")  class Employee(Person):  job\_title=""  def \_\_init\_\_(self):  Person.\_\_init\_\_(self)  print("Employee created")  class Customer(Person):  email=""  def \_\_init\_\_(self):  Person.\_\_init\_\_(self)  print("Customer created")  johnSmith = Person()  janeEmployee = Employee()  bobCustomer = Customer() |

4.1 Проверка пройденного

Создайте следующую программу. Прежде чем приступать к программированию, попробуйте описать её структуру.

Напишите код, описывающий класс **Animal**:

* + Добавьте атрибут имени животного.
  + Добавьте метод **eat()**, выводящий “Ням ням.”
  + Добавьте метод **makeNoise()**, выводящий “[animal name] говорит Гррр.”
  + Добавьте конструктор классу Animal, выводящий “Родилось животное.”

1. Класс **Cat**:
   * Пусть Animal будет его родительским классом.
   * Метод **makeNoise()** класса Cat выводит “[animal name] говорит Мяу.”
   * Конструктор для Cat выводит “Родился кот”, а также вызывает родительский конструктор.
2. Класс **Dog**:
   * Пусть Animal будет его родительским классом.
   * Метод **makeNoise()** для Dog выводит “[animal name] говорит Гав.”
   * Конструктор **Dog** выводит “Родилась собака.”, а также вызывает родительский конструктор.
3. Основная программа:
   * Код, создающий кота, двух собак и одно простое животное.
   * Дайте имя каждому животному.
   * Код, вызывающий **eat()** и **makeNoise()** для каждого животного.

**class** Animal():

name=**''**

grr=**''**

**def** \_\_init\_\_(self, newName, newGrr):

self.name=newName

self.grr=newGrr

**print** (**'born animal'**)

**def** eat(self):

**print** (**'ням-ням'**)

**def** makeNoise(self,newGrr):

**print** (newGrr)

**class** Cat(Animal):

name = **''**

grr = **''**

**def** \_\_init\_\_(self, newName, newGrr):

self.name=newName

self.grr=newGrr

**print**(**'Born cat named'**, newName)

Animal. \_\_init\_\_(self, newName, newGrr)

Animal.eat(self)

Animal.makeNoise(self, newGrr)

**class** Dog(Animal):

name = **''**

grr = **''**

**def** \_\_init\_\_(self, newName, newGrr):

self.name=newName

self.grr=newGrr

**print**(**'Born dog named'**, newName)

Animal. \_\_init\_\_(self, newName, newGrr)

Animal.eat(self)

Animal.makeNoise(self, newGrr)

kot = Cat(**'Murka'**, **'meo'**)

dog = Dog (**'Ryzhyi'**, **'gav'**)

dog = Dog (**'Sonya'**, **'gav'**)

dolfy = Animal(**'Dolfy'**, **'pisk'**)

dolfy.eat()

dolfy.makeNoise(**'pisk'**)