Что такое init в Python? [С примерами]

Описание

Хотите начать работу с объектно-ориентированным дизайном в Python? Сделайте первые шаги уже сегодня, изучив метод __init__ в Python. В этом уроке мы рассмотрим основы классов и объектов Python, а затем перейдем к изучению метода ***__init__***.

К концу этого урока вы сможете ответить на следующие вопросы:

- Что такое переменные экземпляра или атрибуты экземпляра?
- Как метод init помогает инициализировать атрибуты экземпляра?
- Как мы можем установить значения по умолчанию для атрибутов?
- Как мы можем использовать методы класса в качестве конструкторов для создания объектов?

Давайте начнем.

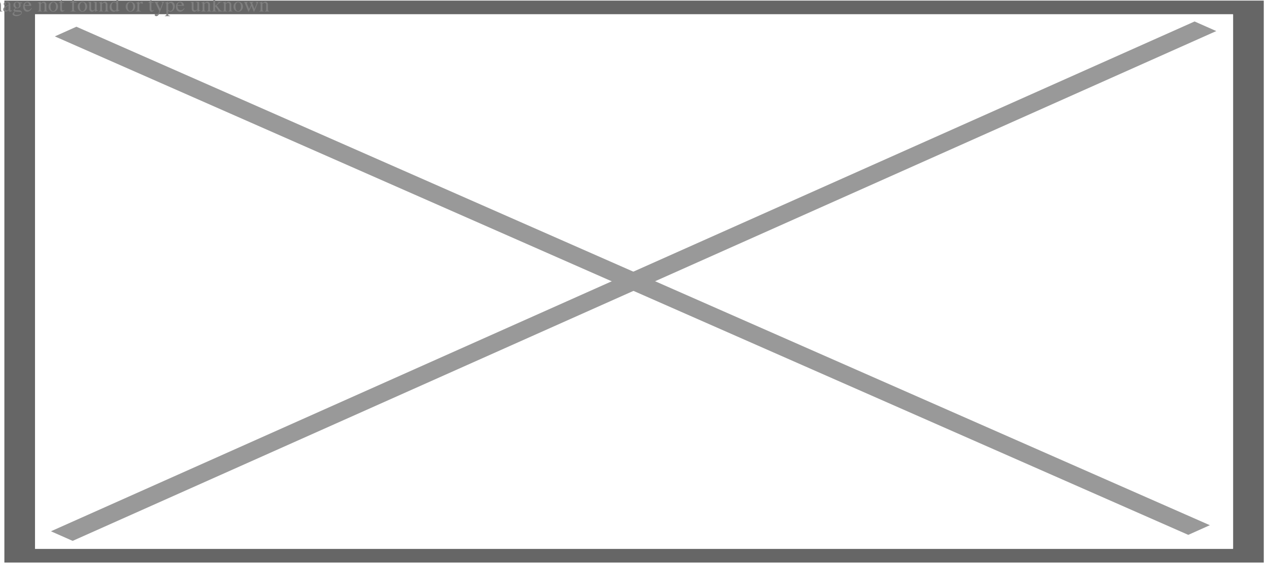
Классы и объекты Python

Классы являются основой объектно-ориентированного программирования в Python. Мы можем создать класс и определить **атрибуты** и **методы**, чтобы связать вместе данные и связанные с ними функциональные возможности. Создав класс, мы можем использовать его как чертеж (или шаблон) для создания объектов (экземпляров). Время примеров! Давайте создадим класс ***Employee***, каждый объект которого будет иметь следующие атрибуты:

- **full_name:** полное имя сотрудника в формате ***firstName lastName***
- **emp_id:** идентификатор сотрудника
- **department:** отдел, к которому он принадлежит
- **experience:** количество лет опыта, который они имеют

Что это значит?

Каждый отдельный сотрудник будет экземпляром или объектом класса ***Employee***. И у каждого объекта будет свое собственное значение для ***full_name***, ***emp_id***, ***department*** и ***experience***. Эти атрибуты также называются переменными экземпляра, и мы будем использовать термины “атрибуты” и “переменные экземпляра” как взаимозаменяемые.



К добавлению атрибутов мы перейдем чуть позже. Пока же мы создадим класс **Employee** следующим образом:

```
class Employee:
    pass
```

Использование **pass** (в качестве заполнителя) помогает нам избежать ошибок при выполнении сценария. Хотя текущая версия класса **Employee** не очень полезна, это все еще действующий класс. Поэтому мы можем создавать объекты класса **Employee**:

```
employee_1 = Employee()

print(employee_1)

#Output: <__main__.Employee object at 0x00FEE7F0>
```

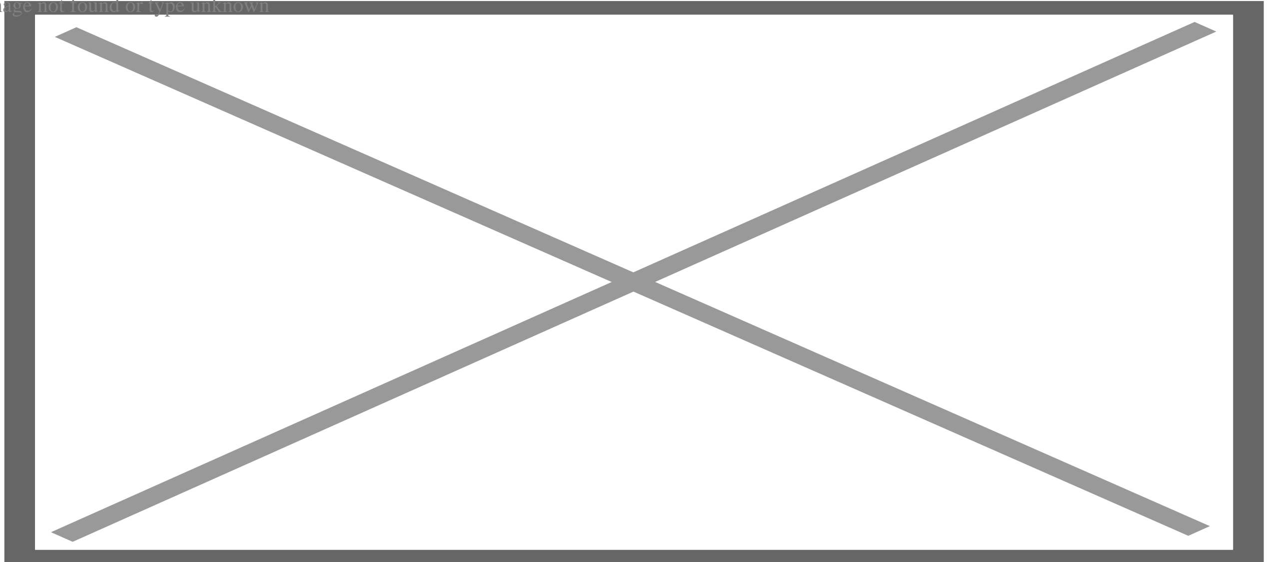
Мы также можем добавить атрибуты и инициализировать их значениями, как показано на рисунке:

```
employee_1.full_name = 'Amy Bell'

employee_1.department = 'HR'
```

Но такой подход к добавлению атрибутов экземпляра неэффективен и чреват ошибками. Кроме того, это не позволяет использовать класс в качестве шаблона для создания объектов. Здесь на помощь приходит метод **__init__**.

Понимание роли метода __init__ в классе Python



Мы должны уметь инициализировать переменные экземпляра при инстанцировании объекта, и метод **__init__** помогает нам в этом. Метод **__init__** вызывается каждый раз, когда создается новый объект класса, чтобы инициализировать значения переменных экземпляра. Если вы программировали на языке вроде C++, вы увидите, что метод __init__ работает аналогично конструкторам.

Определение метода __init__

Давайте добавим метод __init__ в класс Employee:

```
class Employee:
    def __init__(self, full_name,emp_id,department,experience):
        self.full_name = full_name
        self.emp_id = emp_id
        self.department = department

self.experience = experience
```

Параметр self ссылается на экземпляр класса, а self.attribute инициализирует атрибут экземпляра значением, указанным в правой части. Теперь мы можем создавать объекты следующим образом:

```
employee_2 = Employee('Bella Joy','M007','Marketing',3)
print(employee_2)

# Output: <__main__.Employee object at 0x017F88B0>
```

Когда мы распечатываем объекты сотрудников, мы не получаем никакой полезной информации, кроме класса, к которому они принадлежат. Давайте добавим метод **__repr__**, который определяет строку представления для класса:

```
 def __repr__(self):
        return f"{self.full_name},{self.emp_id} from {self.department} with {s
```

Добавив **__repr__** к классу **Employee**, мы получим:

```
class Employee:
    def __init__(self, full_name,emp_id,department,experience):
        self.full_name = full_name
        self.emp_id = emp_id
        self.department = department
        self.experience = experience

     def __repr__(self):
        return f"{self.full_name},{self.emp_id} from {self.department} with {s
```

Теперь у объектов сотрудников есть полезная строка представления:

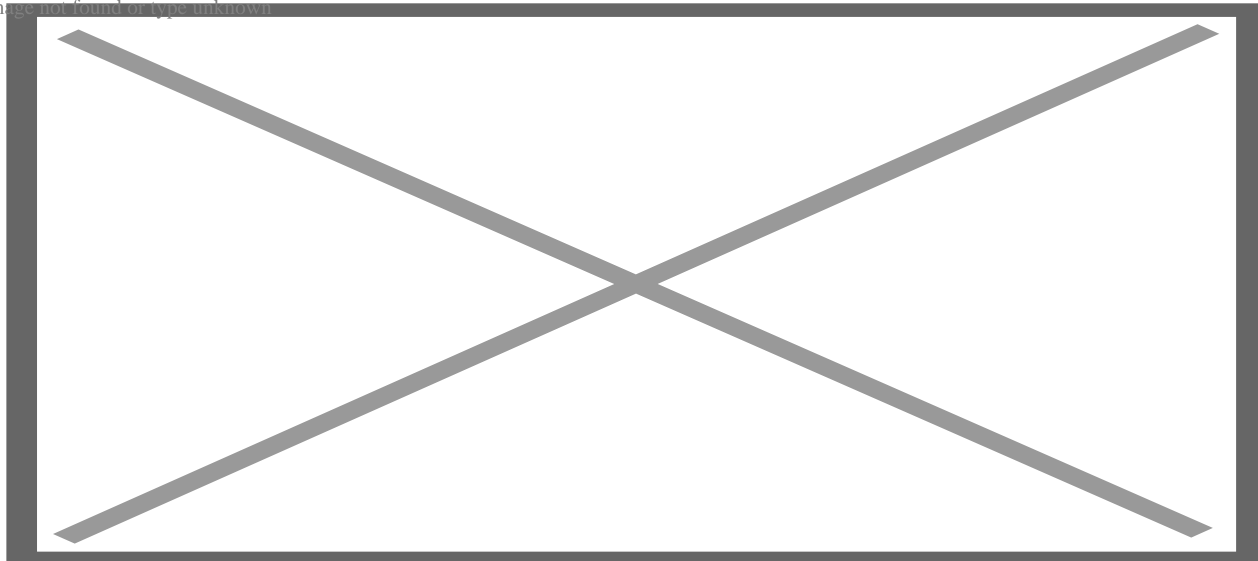
```
print(employee_2)
# Output: Bella Joy,M007 from Marketing with 3 years of experience.
```

Некоторые условные обозначения

Прежде чем мы продолжим, сделаем несколько замечаний:

- Мы использовали self в качестве первого параметра в методе __init__ для ссылки на сам экземпляр класса и использовали self.attribute_name для инициализации различных атрибутов. Использование self является предпочтительным (однако вы можете использовать любое другое имя).
- При определении метода __init__ мы установили имена параметров в определениях __init__ в соответствии с именами атрибутов. Это улучшает читабельность.

Как добавить значения по умолчанию для атрибутов



В примере, который мы приводили до сих пор, все атрибуты являются обязательными. То есть создание объекта будет успешным, только если мы передадим конструктору значения для всех полей. Попробуйте создать объект класса Employee без передачи значения атрибута experience:

```
employee_3 = Employee('Jake Lee','E001','Engineering')
```

Вы получите следующую ошибку:

```
Traceback (most recent call last):
  File "main.py", line 22, in <module>
    employee_3 = Employee('Jake Lee','E001','Engineering')

TypeError: __init__() missing 1 required positional argument: 'experience'
```

Но если вы хотите сделать определенные атрибуты необязательными, вы можете сделать это, предоставив значения по умолчанию для этих атрибутов при определении метода **init**. Здесь мы задаем значение по умолчанию 0 для атрибута experience:

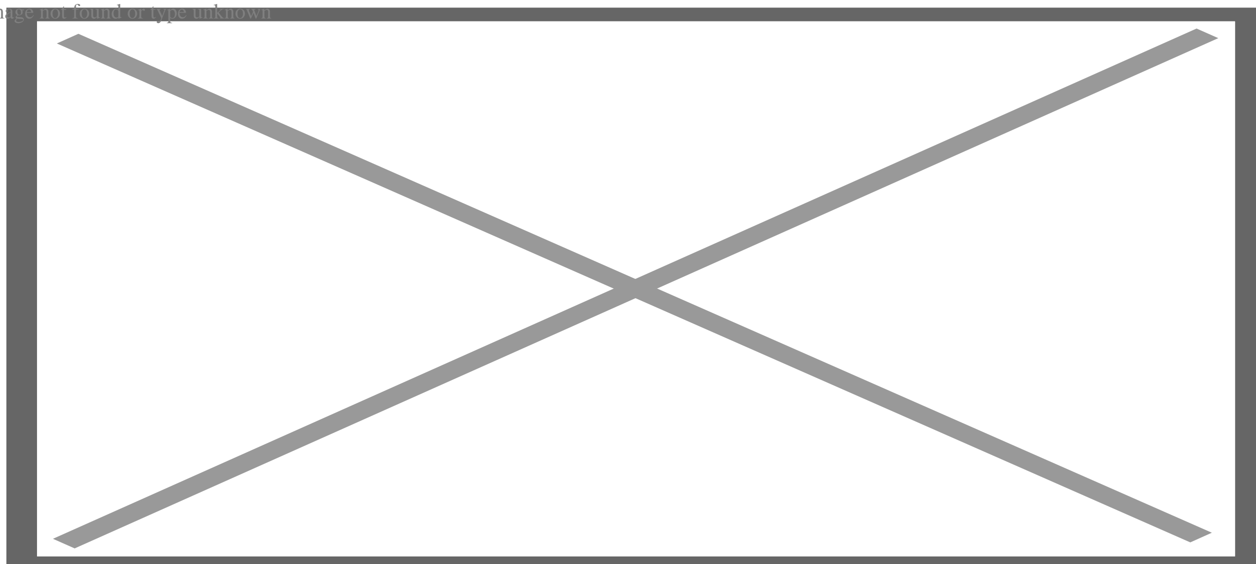
```
class Employee:
    def __init__(self, full_name,emp_id,department,experience=0):
        self.full_name = full_name
        self.emp_id = emp_id
        self.department = department
        self.experience = experience
```

```
    def __repr__(self):
       return f"{self.full_name},{self.emp_id} from {self.department} with {s
```

Объект employee_3 создается без значения для атрибута experience; по умолчанию для experience используется значение 0.

```
employee_3 = Employee('Jake Lee','E001','Engineering')
print(employee_3.experience)
#
Output: 0
```

Альтернативные конструкторы классов с использованием методов классов



До сих пор мы видели только, как определить метод __init__ и установить значения по умолчанию для атрибутов, когда это необходимо. Мы также знаем, что в конструкторе нужно передавать значения для необходимых атрибутов. Однако иногда значения для переменных экземпляра (или атрибутов) могут быть доступны в другой структуре данных, например, в кортеже, словаре или строке JSON.

Что же нам делать?

Давайте рассмотрим пример. Предположим, у нас есть значения переменной экземпляра в словаре Python:

```
dict_fanny = {'name':'Fanny Walker','id':'H203','dept':'HR','exp':2}
```

Мы можем обратиться к словарю и получить все атрибуты следующим образом:

```
name = dict_fanny['name']
id = dict_fanny['id']
dept = dict_fanny['dept']

exp = dict_fanny['exp']
```

После этого вы можете создать объект, передав эти значения в конструктор класса:

```
employee_4 = Employee(name, id, dept, exp)
print(employee_4)

# Output: Fanny Walker,H203 from HR with 2 years of experience.
```

Помните: это нужно делать для каждого нового объекта, который вы создаете. Такой подход неэффективен, и мы определенно можем сделать лучше. Но как? В Python мы можем использовать методы класса в качестве конструкторов для создания объектов класса. Для создания метода класса мы используем декоратор ***@classmethod***. Давайте определим метод, который анализирует словарь, получает значения переменных экземпляра и использует их для создания объектов ***Employee***
.

```
    @classmethod
    def from_dict(cls,data_dict):
        full_name = data_dict['name']
        emp_id = data_dict['id']
        department = data_dict['dept']
        experience = data_dict['exp']

return cls(full_name, emp_id, department, experience)
```

Когда нам нужно создать объекты, используя данные из словаря, мы можем использовать метод класса ***from_dict()***. Обратите внимание на использование cls в методе класса вместо self. Точно так же, как мы используем self для ссылки на экземпляр, cls используется для ссылки на класс. Кроме того, методы класса привязываются к классу, а не к объектам. Поэтому, когда мы вызываем метод класса **from_dict()** для создания объектов, мы вызываем его на классе **Employee**:

```
emp_dict = {'name':'Tia Bell','id':'S270','dept':'Sales','exp':3}
employee_5 = Employee.from_dict(emp_dict)
print(employee_5)

# Output: Tia Bell,S270 from Sales with 3 years of experience.
```

Теперь, если у нас есть словарь для каждого из n сотрудников, мы можем

использовать метод класса from_dict() в качестве конструктора для инстанцирования объектов – без необходимости получать значения мгновенных переменных из словаря.

Заметка о переменных класса

Здесь мы определили метод класса, который привязан к классу, а не к отдельным экземплярам. Подобно методам класса, мы также можем иметь переменные класса. Как и методы класса, переменные класса привязаны к классу, а не к экземпляру. Когда атрибут принимает фиксированное значение для всех экземпляров класса, мы можем определить их как переменные класса.

Часто задаваемые вопросы

Зачем нужен метод __init__ в Python?

Метод __init__ в определении класса позволяет нам инициализировать атрибуты или переменные экземпляра для всех экземпляров класса. Метод __init__ вызывается каждый раз, когда создается новый экземпляр класса.

Можно ли иметь несколько методов __init__ в классе Python?

Цель наличия нескольких методов __init__ в классе Python – предоставить несколько конструкторов для создания объектов. Но вы не можете определить несколько методов __init__. Если вы определите несколько методов __init__, вторая и самая последняя реализация перезапишет первую. Однако вы можете использовать декоратор @classmethod для определения методов класса, которые могут быть использованы в качестве конструкторов для инстанцирования объектов.

Что произойдет, если не определить метод __init__ в классе?

Если вы не определите метод __init__, вы все равно сможете инстанцировать объекты. Однако вам придется вручную добавлять переменные экземпляра и присваивать значения каждой из них. Вы не сможете передать значения переменных экземпляра в конструкторе. Это не только чревато ошибками, но и противоречит цели использования класса в качестве чертежа, на основе которого можно создавать объекты.

Можно ли задать значения по умолчанию для аргументов в методе __init__?

Да, можно задать значения по умолчанию для одного или нескольких атрибутов при определении метода __init__. Предоставление значений по умолчанию помогает сделать эти атрибуты необязательными в конструкторе. Атрибуты принимают значения по умолчанию, когда мы не передаем значения этих атрибутов в конструкторе.

Можно ли изменять атрибуты вне метода __init__?

Да, вы всегда можете обновить значение атрибута вне метода __init__. Вы также можете динамически добавлять новые атрибуты к экземпляру после создания конкретного экземпляра.

Заключение

В этом руководстве мы узнали, как использовать метод **__init__** для инициализации значений переменных экземпляра. Хотя эта процедура проста, она может быть повторяющейся – особенно если у вас много атрибутов. Если вам интересно, вы можете изучить модуль dataclasses. В Python 3.7 и более поздних версиях вы можете использовать встроенный модуль классов данных для создания классов данных, которые хранят данные. В дополнение к реализации по умолчанию **__init__** и других часто используемых методов, они поставляются с множеством классных функций для подсказок типов, сложных значений по умолчанию и оптимизации.

Дата Создания
22.06.2023