Материалы занятия

Курс: Разработка Web-приложений на Python, с применением Фреймворка Django
Дисциплина: Основы программирования на Python

Тема занятия №15: ООП. Полиморфизм

1. Введение

Полиморфизм в объектно-ориентированном программировании – это возможность обработки разных типов данных, т. е. принадлежащих к разным классам, с помощью "одной и той же" функции, или метода. На самом деле одинаковым является только имя метода, его исходный код зависит от класса. Кроме того, результаты работы одноименных методов могут существенно различаться. Поэтому в данном контексте под полиморфизмом понимается множество форм одного и того же слова – имени метода.

2. Полиморфизм оператора сложения

Мы знаем, что оператор + часто используется в программах на Python. Но он не имеет единственного использования.

Для целочисленного типа данных оператор + используется чтобы сложить операнды.

```
num1 = 1
num2 = 2
print(num1 + num2)
```

Итак, программа выведет на экран 3.

Подобным образом оператор + для строк используется для конкатенации.

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

В результате будет выведено Python Programming.

Здесь мы можем увидеть единственный оператор + выполняющий разные операции для различных типов данных. Это один из самых простых примеров полиморфизма в Python.

3. Полиморфизм на примере функции len()

```
print(len("Programiz"))
print(len(["Python", "Java", "C"]))
print(len({"Name": "John", "Address": "Nepal"}))
```

Вывод:

9
3
2

Здесь мы можем увидеть, что различные типы данных, такие как строка, список, кортеж, множество и словарь могут работать с функцией len(). Однако, мы можем увидеть, что она возвращает специфичную для каждого типа данных информацию.

4. Полиморфизм в классах

Рассмотрим пример полиморфизма на методе, который перегружает функцию str(), которую автоматически вызывает функция print().

Если вы создадите объект собственного класса, а потом попробуете вывести его на экран, то получите информацию о классе объекта и его адрес в памяти. Такое поведение

функции str() по-умолчанию по отношению к пользовательским классам запрограммировано на самом верхнем уровне иерархии, где-то в суперклассе, от которого неявно наследуются все остальные.

```
class A:
   def __init__(self, v1, v2):
      self.field1 = v1
      self.field2 = v2


a = A(3, 4)
b = str(a)
print(a)
print(b)
```

Вывод:

```
<__main__.A object at 0x7f251ac2f8d0>
<__main__.A object at 0x7f251ac2f8d0>
```

Здесь мы используем переменную b, чтобы показать, что функция print() вызывает str() неявным образом, так как вывод значений обоих переменных одинаков.

Если же мы хотим, чтобы, когда объект передается функции print(), выводилась какая-нибудь другая более полезная информация, то в класс надо добавить специальный метод __str__(). Этот метод должен обязательно возвращать строку, которую будет в свою очередь возвращать функция str(), вызываемая функций print():

```
class A:
   def __init__(self, v1, v2):
      self.field1 = v1
      self.field2 = v2

   def __str__(self):
      s = str(self.field1)+" "+str(self.field2)
      return s


a = A(3, 4)
b = str(a)
print(a)
print(b)
```

Вывод:

```
3 4
3 4
```

Какую именно строку возвращает метод __str__(), дело десятое. Он вполне может строить квадратик из символов:

```
class Rectangle:
   def __init__(self, width, height, sign):
      self.w = int(width)
      self.h = int(height)
      self.s = str(sign)
   def __str__(self):
      rect = []
      # количество строк
      for i in range(self.h):
         # знак повторяется w раз
```

```
            rect.append(self.s * self.w)
        # превращаем список в строку
        rect = '\n'.join(rect)
        return rect

b = Rectangle(10, 3, '*')
print(b)
```
Вывод:

```
**********
**********
**********
```
Рассмотрим следующий пример:
```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a cat. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Meow")
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print(f"I am a dog. My name is {self.name}. I am {self.age} years old.")

    def make_sound(self):
        print("Bark")

cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```
Вывод:
```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```
Здесь мы создали два класса Cat и Dog. У них похожая структура и они имеют методы с одними и теми же именами info() и make_sound().

Однако, заметьте, что мы не создавали общего класса-родителя и не соединяли классы вместе каким-либо другим способом. Даже если мы можем упаковать два разных объекта в

кортеж и итерировать по нему, мы будем использовать общую переменную animal. Это возможно благодаря полиморфизму.

5. Утиная типизация

Полиморфизм без наследования в форме утиной типизации, доступной в Python, благодаря его динамической системе типирования. Это означает, что до тех пор, пока классы содержат одинаковые методы, интерпретатор Python не различает их, поскольку единственная проверка вызовов происходит во время выполнения.

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")
    def feathers(self):
        print("The duck has white and gray feathers.")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")

def in_the_forest(obj):
    obj.quack()
    obj.feathers()

donald = Duck()
john = Person()
in_the_forest(donald)
in_the_forest(john)
```

Выход:

Quaaaaaack! \ У утки белые и серые перья. \ Человек подражает утке. \ Человек берет перо с земли и показывает его.