Полиморфизм в Python

[Автор оригинала: Parewa Labs Pvt. Ltd.](https://www.programiz.com/python-programming/polymorphism)

* [Python](https://habr.com/ru/hub/python/),
* [Программирование](https://habr.com/ru/hub/programming/)
* Перевод

В этой статье мы изучим полиморфизм, разные типы полиморфизма и рассмотрим на примерах как мы можем реализовать полиморфизм в Python.

Что такое полиморфизм?

В буквальном значении полиморфизм означает множество форм.

Полиморфизм — очень важная идея в программировании. Она заключается в использовании единственной сущности(метод, оператор или объект) для представления различных типов в различных сценариях использования.

Давайте посмотрим на пример:

Пример 1: 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.

Полиморфизм функций

В Python есть некоторые функции, которые могут принимать аргументы разных типов.

Одна из таких функций — len(). Она может принимать различные типы данных. Давайте посмотрим на примере, как это работает.

Пример 2: полиморфизм на примере функции len()

print(len("Programiz"))

print(len(["Python", "Java", "C"]))

print(len({"Name": "John", "Address": "Nepal"}))

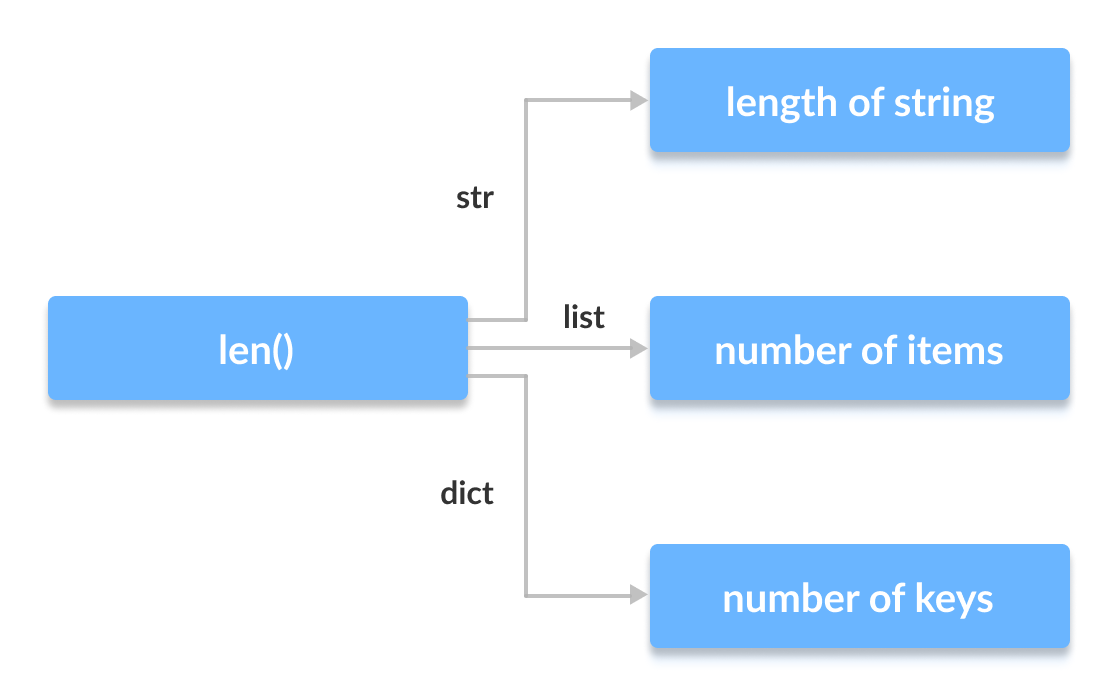
**Вывод:**

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.

Полиморфизм функции len()

Полиморфизм в классах

Полиморфизм — очень важная идея в объектно-ориентированном программировании.

Мы можем использовать идею полиморфизма для методов класса, так как разные классы в Python могут иметь методы с одинаковым именем.

Позже мы сможем обобщить вызов этих методов, игнорируя объект, с которым мы работаем. Давайте взглянем на пример:

Пример 3: полиморфизм в методах класса

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.

Полиморфизм и наследование

Как и в других языках программирования, в Python дочерние классы могут наследовать методы и атрибуты родительского класса. Мы можем переопределить некоторые методы и атрибуты специально для того, чтобы они соответствовали дочернему классу, и это поведение нам известно как **переопределение метода(method overriding)**.

Полиморфизм позволяет нам иметь доступ к этим переопределённым методам и атрибутам, которые имеют то же самое имя, что и в родительском классе.

Давайте рассмотрим пример:

Пример 4: переопределение метода

from math import pi

class Shape:

def \_\_init\_\_(self, name):

self.name = name

def area(self):

pass

def fact(self):

return "I am a two-dimensional shape."

def \_\_str\_\_(self):

return self.name

class Square(Shape):

def \_\_init\_\_(self, length):

super().\_\_init\_\_("Square")

self.length = length

def area(self):

return self.length\*\*2

def fact(self):

return "Squares have each angle equal to 90 degrees."

class Circle(Shape):

def \_\_init\_\_(self, radius):

super().\_\_init\_\_("Circle")

self.radius = radius

def area(self):

return pi\*self.radius\*\*2

a = Square(4)

b = Circle(7)

print(b)

print(b.fact())

print(a.fact())

print(b.area())

**Вывод:**

Circle

I am a two-dimensional shape.

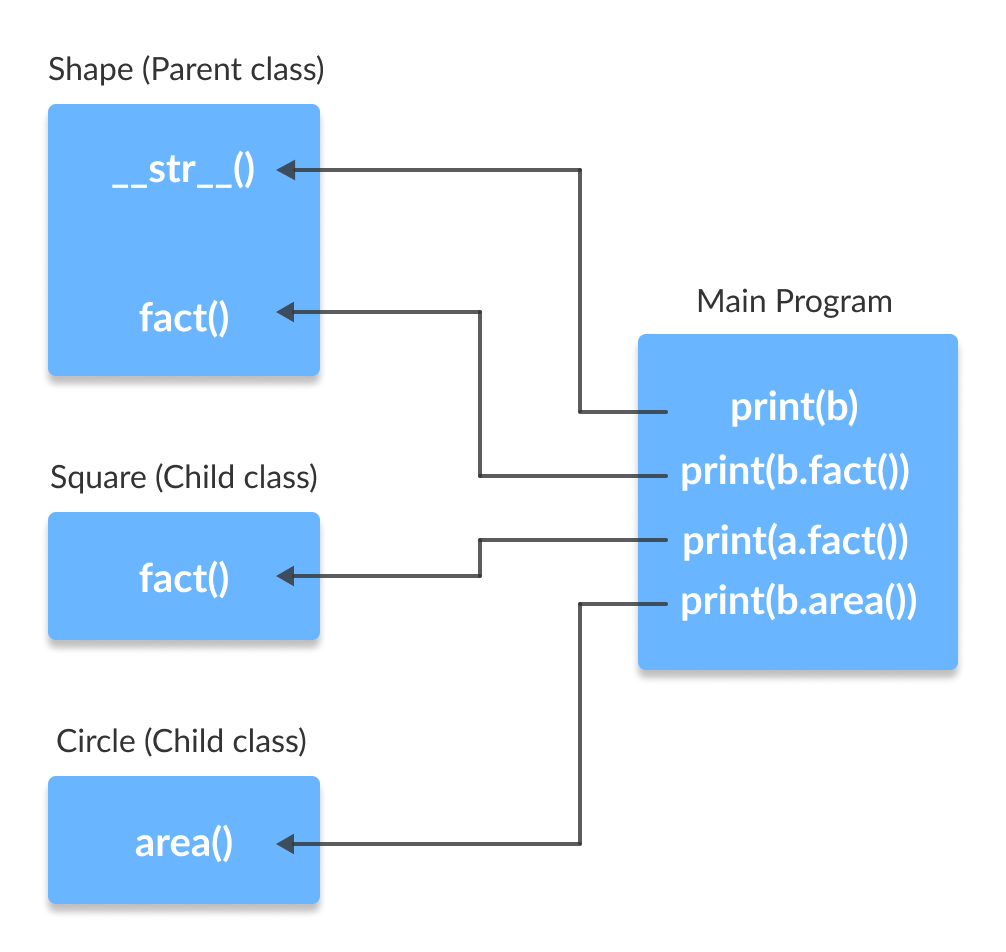
Squares have each angle equal to 90 degrees.

153.93804002589985

Здесь мы можем увидеть, что такие методы как \_\_str\_\_(), которые не были переопределены в дочерних классах, используются из родительского класса.

Благодаря полиморфизму интерпретатор питона автоматически распознаёт, что метод fact() для объекта a(класса Square) переопределён. И использует тот, который определён в дочернем классе.

С другой стороны, так как метод fact() для объекта b не переопределён, то используется метод с таким именем из родительского класса(Shape).



Полиморфизм на примере дочерних и родительских классов в питоне

Заметьте, что **перегрузка методов(method overloading)**— создание методов с одним и тем же именем, но с разными типами аргументов не поддерживается в питоне.