DOI:10.30546/gdu.2023.12-20

RİYAZİYYAT VƏ MEXANİKA

UOT:004.3

PYTHONDA VARİSLİK VƏ POLİMORFİZM

Babayeva Şəkər Maarif qızı
Gəncə Dövlət Universiteti
shakarbabayeva@mail.ru

Xülasə: *Varislik mövcud olan sinif əsasında yenisinin yaradılması imkanına aiddir. Bununla varis sinfi əcdad sinfi ilə eyni atribut və metodları ehtiva edəcək, lakin yeni metodlar və atributlar əlavə etməklə genişləndirilə bilər (və genişləndirilməlidir).*

Polimorfizm Python-da obyekt yönümlü proqramlaşdırmanın mühüm elementidir. Polimorfizm Python-u daha çox universal və səmərəli dil edir. Bu, birdən çox işi yerinə yetirmək üçün bir ifadə və ya funksiyadan istifadə etməyə imkan verir. O, həmçinin metodları yenidən təyin etməklə müxtəlif siniflərin unikal məqsədləri üçün metod adlarını təkrar istifadə etməyə imkan verir.

Cari məqalə Pythonda OOP-nın varislik və polimorfizm prinsipləri və onlardan istifadə etməklə proqramların yaradılmasına həsr edilmişdir. Məqalədə bir neçə oxşar qrup obyektlərin olmasını tələb edən məsələlər üzərində varislik və polimorfizm adlanan yanaşmaların tətbiqi əlverişli olduğu izah edilir.

Açar sözlər: sinif, obyekt, xassələr, metodlar, varislik, irsi ierarxiyası, polymorfizm, metodların yenidən təsvir (təyin) edilməsi.

Ключевые слова: класс, объект, свойства, методы, наследование, иерархия наследования, полиморфизм, переопределение методов.

Key words: class, oject, properties, methods, inheritance, inheritance hierarchy, polymorphism, method overriding

Obyekt yönümlü proqramlaşdırmanın (OOP) dörd əsas prinsipləri bunlardır:

- Abstraksiya
- İnkapsulyasiya
- Varislik (miras, irsiyyət)
- Polimorfizm

Varislik, bir sinfin digərinin atribut və metodlarını miras alması prosesidir. Xassələri və metodları miras qalan sinfə Valideyn və ya Supersinif deyilir. Xassələri və metodları miras alan sinfə – varis sinif və ya alt sinif deyilir. Sıfırdan başlamaq əvəzinə, mövcud olan sinifə əsaslanaraq yeni sinif yarada bilərsiniz. Yeni sinif adından sonra mötərizədə ana (valideyn) sinfi göstərmək lazımdır. Varis sinfi öz ana sinifinin atributlarını miras alır. Siz bu atributları törəmə sinifdə müəyyən edilmiş kimi istifadə edə bilərsiniz Varis sinifdə valideyn sinfinin xassələrini və metodlarını yenidən təyin (təsvir) etmək olar (bu polimorfizmdə istifadə olunur).

Nə üçün miras lazım ola bilər?

Tutaq ki, proqramımızda elan edilmiş iki müstəqil sinifimiz var:

```
class Geom:
 name = 'Geom'
 class Line:
 def draw(self):
 print("Xəttin çəkilməsi")
```

DOI:10.30546/gdu.2023.12 -20

Bildiyimiz kimi, indi **Geom** sinifinin bir nümayəndəsini yaratsaq:

```
g = Geom()
```

onda name xassəsinə daxil ola bilərik:

```
print(g.name)
```

Lakin biz Line sinfinin draw() metodunu çağıra bilməyəcəyik, çünki o, tamamilə fərqli, müstəqil sinifdir. Line obyekti üçün də onun draw() metodunu çağıra bilərsiniz, lakin başqa Geom sinifinin name xassəsinə daxil ola bilməzsiniz:

```
l = Line()
l.draw()
```

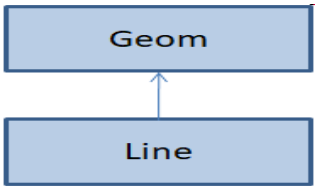
Yəni bu siniflər, onların adlar sahələri bir-birindən tam müstəqildir. Lakin, lazım gələrsə, biz onlar arasında əlaqə yarada və məsələn, Geom sinifinin ictimai atributlarını Line sinfində mövcud edə bilərik. Belə bir əlaqəyə keçid aşağıdakı kimi yazılır:

```
class Line(Geom):
    def draw(self):
        print("Xəttin çəkilməsi")
```

Mötərizədə Line sinfi ilə genişləndirəcəyimiz Geom sinfini qeyd etdik və Geom sinfində mövcud olan hər şey avtomatik olaraq Line sinfində əlçatan olur. Xüsusilə, indi Line sinifinin obyekti vasitəsilə name xassəsinə təhlükəsiz şəkildə daxil ola bilərik:

```
print(l.name)
```

Əgər bir sinif digəri əsasında müəyyən edilirsə, belə konstruksiya miras adlanır. Bundan əlavə, Geom sinfi ana və ya əsas sinif, Line sinfi isə ana (valideyn) sinfinin alt sinfi və ya uşaq (varis) sinfi adlanır. Qrafik olaraq, siniflərin irsi iyerarxiyası adətən oxlarla birləşdirilən düzbucaqlılar kimi təsvir edilir və ox varisdən əcdad sinfə yönəldilir:



İndi xətt obyektinə koordinatları yazmaq üçün Line sinfinə set_coords metodunu əlavə edək:

```
def set_coords(self, x1, y1, x2, y2):
    self.x1 = x1
    self.y1 = y1
    self.x2 = x2
    self.y2 = y2
```

Və sonra, düzbucaqlılar üçün başqa oxşar sinif təyin edək:

```
class Rect(Geom):
    def set_coords(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
    def draw(self):
        print("Düzbucaqlının çəkilməsi")
```

Gördüyünüz kimi, burada kodun təkrarlanması mövcuddur, və müxtəlif həndəsi fiqurlar üçün siniflərin artması ilə bu kod sürətlə böyüyəcəkdir. Bunun qarşısını almaq üçün biz bütün varis (Geom-dan) sinifləri üçün ümumi olanları baza sinfinə köçürə bilərik və

DOI:10.30546/gdu.2023.12 -20

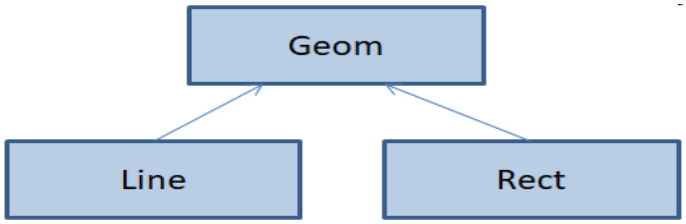
xüsusən də orada set_coords metodunu yaza bilərik. Nəticədə siniflərin aşağıdakı təsvirlərini alırıq:

```
class Geom:
    name = 'Geom'
    def set_coords(self, x1, y1, x2, y2):
        self.x1 = x1
        self.y1 = y1
        self.x2 = x2
        self.y2 = y2
class Line(Geom):
    def draw(self):
        print("Xəttin çəkilməsi")
class Rect(Geom):
    def draw(self):
        print("Düzbucaqlının çəkilməsi")
```

Bunun nəticəsində, kodun təkrarlanmasını aradan qaldırdıq və varis siniflərin obyektləri vasitəsilə baza sinifindən set_coords metodunu çağıra bilərik:

```
l = Line()
r = Rect()
l.set_coords(1, 1, 2, 2)
r.set_coords(1, 1, 2, 2)
```

İndi miras iyerarxiyası belə olacaq:



İstifadəçi tərəfindən yaradılan sinif avtomatik olaraq Pythonun baza obyekt sinifindən miras alır. Məsələn, heç bir atributları olmayan bir Geom sinfi yaratsaq:

```
class Geom:
    pass
```

İDLE mühitində aşağıda "Geom." yazaraq, müxtəlif metodların və xassələrin siyahısını görəcəyik. Məsələn, __class__ xassəsinin qiymətini görmək olar:

```
print(Geom.__class__)
<class 'type'>
```

Təxmin etmək olar ki, bunlar **object** baza sinfində var və ondan miras alınır (Python 3-dən bəri əlavə olunub). Bu, aşağıdakılara ekvivalentdir:

```
class Geom(object):
    pass
```

Bu nə üçün edilib? Aydındır ki, siniflərlə işləmək üçün standart baza funksionallığı təmin etmək üçün. Konkret olaraq, sinfin nümayəndəsini yaratdıqda:

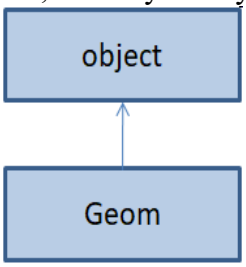
```
g = Geom()
```

və sonra onu print( ) funksiyası ilə ekrana xaric edəndə:

```
print(g)
<__main__.Geom object at 0x000001F4AEA7A950>  # Nəticə avtomatik olaraq object baza sinfində təsvir olunmuş sehrli metod __str__ icra olunur. Və bütün atributlarla uyğun
```

DOI:10.30546/gdu.2023.12-20

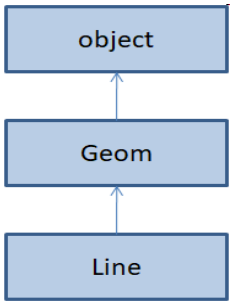
prosess baş verir. Təsəvvür edə bildiyiniz kimi, bu cür baza funksionallığa sahib olmaq çox rahatdır, buna görə də Geom susmaqla **obyekt** baza sinfindən bu şəkildə miras qalır. Nəticədə, miras iyerarxiyası belə olur:



Bununla belə, Geom-un varis sinfini əlavə etsək, məsələn, xətti təsvir edən Line sinifini:

```
class Geom(object):
    pass
class Line(Geom):
    pass
```

varislik ieraxiyası aşağıdakı kimi olacaq:



Təbii ki, burada Line obyektləri də obyekt sinfinin bütün ictimai atributlarına tam giriş imkanına malikdir:

```
l = Line()
print(l.__class__)
```

Bundan əlavə, müəyyən bir sinfin başqa bir sinfin alt sinfi olub olmadığını müəyyən edə bilərik. Bu **issubclass()** funksiyası tərəfindən edilir, məsələn:

```
print(issubclass(Line, Geom))
```

Line sinfi Geom sinfinin alt sinfidirsə, True qaytarır. Ancaq onları fərqli bir ardıcıllıqla göstərsəniz:

```
print(issubclass(Geom, Line))
```

onda biz False alırıq, çünki Geom sinfi Line sinfinin varisi deyil.

Lakin bu funksiya sinif obyektləri ilə işləmir. Belə yazsaq:

```
print(issubclass(l, Geom))
```

onda biz səhv alacayıq, çünki funksiyanın birinci arqumenti (l) sinfin nümayəndəsi yox, sinif olmalıdır.

Əgər obyektin müəyyən bir sinifə, o cümlədən baza sinfinə aid olub-olmadığını yoxlamaq lazımdırsa, onda **isinstance()** funksiyasından istifadə etməliyik:

```
print(isinstance(l, Geom))
print(isinstance(l, Line))
```

Hər iki halda biz True qiymətini alırıq. Həmçinin, baza sinif obyekti üçün yoxlama doğru olacaq:

DOI:10.30546/gdu.2023.12-20

```
print(isinstance(l, object))
True  # Nəticə
```

Bu, bir daha göstərir ki, bütün siniflər objekt sinfindən miras alır (susmaqla).

Beləliklə, iki sinif və nümayəndənin əlaqəsini yoxlamaq üçün issubclass() və ya isinstance() funksiyalarından istifadə edilir.

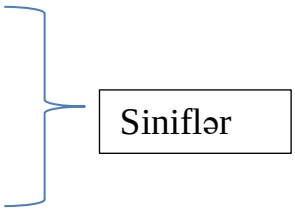
Əgər verilmiş alt sinif (alt) həqiqətən sup sinfininın alt sinfidirsə, məntiqi funksuya issubclass(alt, sup) True qaytarır. Bu funksiyanın arqumentləri ancaq siniflər ola bilər.

Əgər **obj** sinif nümayəndəsi və ya alt sinfin nümayəndəsidirsə, məntiqi funksiya isinstance(obj, Class) True qaytarır. Bu funksiya siniflərlə və onların nümayəndələri ilə işləyir.

Daxili (bielt-in – quraşdırılmış) verilənlər tiplərindən miras.

Python dilinin maraqlı faktı ondan ibarətdir ki, bütün standart verilənlər tipləri siniflərdir:

int
float
list
dict
tuple
set
...



Əgər onlar üçün issubclass() funksiyasını yerinə yetirsək, bunu asanlıqla yoxlaya bilərik:

```
issubclass(int, object)
issubclass(list, object)
```

Hamısının nəticəsi True olacaq və biz bilirik ki, bu funksiya obyektlərlə deyil, yalnız siniflərlə işləyir, ona görə də bu fakt təsdiq edir ki, bu tiplər Python dilində siniflərdir.

Metodların yenidən təyin edilməsinə nümunə:

```
class Parent:  # valideyn sinfinin elan edilməsi
    def my_method(self):
        print('Valideyn sinfinin çağırılması')
class Child(Parent):  # Varis sinfinin elan edilməsi
    def my_method(self):
        print('Varis sinfinin çağırılması')
c = Child()  #  Child sinfinin nümayəndəsinin yaradılması
c.my_method()  # bu metod varis sinfində yenidən təyin edilib
```

Bu kod icra olunanda aşağıdakı nəticə alınır:

```
Varis sinfinin çağırılması
```

Metodların yenidən təyin edilməsilə tanış olandan sonra polimorfizmin öyrənilməsinə keçə bilərik.

Python öz universallığı və istifadəçi üçün rahat sintaksisi sayəsində ən populyar proqramlaşdırma dillərindən biridir. Obyekt yönümlü proqramlaşdırmanı (OOP) dəstəkləyən digər dillər kimi Python da polimorfikdir. Polimorfizm Python-da obyekt yönümlü proqramlaşdırmanın mühüm elementidir. Polimorfizm Python-u daha çox universal və səmərəli dil edir. Bu, birdən çox işi yerinə yetirmək üçün bir ifadə və ya funksiyadan istifadə etməyə imkan verir. O, həmçinin metodları yenidən təyin etməklə müxtəlif siniflərin unikal məqsədləri üçün metod adlarını təkrar istifadə etməyə imkan verir.

DOI:10.30546/gdu.2023.12-20

Python dinamik tipli dildir və onda polimorfizm hər yerdə özünü birüzə verir. Əslində, Python dilində bütün əməliyyatlar polimorfikdir: xaric etmə, elementin alınması, * operatoru və bir çox başqaları. Python-da əməliyyatın sintaktik mənasını təyin edən obyektlərdir. Məsələn, * operatoru yalnız emal ediləcək obyektlər üçün (göstərişdir) instruksiyadır. Əməliyyatın mənası emal olunan obyektlərin tiplərindən asılıdır.

Polimorfizm - müxtəlif siniflərdə eyni metodun fərqli davranışıdır (icrasıdır). Məsələn, iki ədədin cəmini və iki sətrin cəmini tapa bilərik. Bu halda biz fərqli nəticə əldə edirik, çünki rəqəmlər və sətirlər müxtəlif siniflərdir.

```
>>> 1 + 1
2    # toplama
>>> "1" + "1"
'11'  # konkatenasiya
```

Pythonda aşağıdakı kod səhv olmayacaq:

```
def f(x):
    return x * 2
print(f(1) * f("2"))
print(f(3.) * f(f(4)))
```

Parametrlərin tipləri üzrə funksiyaların polimorfizmi sayəsində arqumentlərin hər birilə özünəməxsus əməliyyat aparılacaq: ədədlər ikiyə vurulacaq və sətir iki dəfə təkrarlanacaq.

Pythonda aşağıdakı kod da səhv olmayacaq:

```
def f(x):
    return x
def f(x):
    return 2 * x
def f(x):
    return 3 * x
print(f(5))
```

Bu halda, ilk iki funksiyanın, onların adları artıq adlar sahəsində mövcud olmadığından, əslində mövcudluğu ləğv edilir. print(f(5)) funksiyası çağırılan zaman sonuncu funksiya çağırılacaq (çünki funksiyanın yenidən təyin (təsvir) edilməsi baş verir), hesablamanın nəticəsi 15 olacaq.

Əvvəl qeyd olunduğu kimi, OOP daxilində polimorfizm adətən varisin sinfində əsas sinfin metodlarını yenidən təyin edilməsi nöqteyi-nəzərindən istifadə olunur. Bunu görməyin ən asan yolu bir nümunədir.

Iki sinif (Rectangle və Square) yaradaq:

```
class Rectangle:
    def __init__(self, w, h):
        self.w = w
        self.h = h
    def get_rect_pr(self):
        return 2*(self.w+self.h)
class Square:
    def __init__(self, a):
        self.a = a
    def get_sq_pr(self):
        return 4*self.a
```

DOI:10.30546/gdu.2023.12-20

Bu siniflərdə düzbucaqlı və kvadratın perimetrlərini hesablamaq üçün get_rect_pr() və get_sq_pr() funksiyaları təsvir edilib. Bu siniflərin nümayəndələrini yaradaq və tərəflərin qiymətlərini daxil edərək perimetrlərin qiymətlərini ekrana xariç edək:

```
r1 = Rectangle(1, 2)
r2 = Rectangle(3, 4)
print(r1.get_rect_pr(), r2.get_rect_pr())
 s1 = Square(10)
s2 = Square(20)
print(s1.get_sq_pr(), s2.get_sq_pr())
```

Aşağıdakı nəticəni alırıq:

```
6 14
40 80
```

Bütün obyektləri kolleksiyaya (siyahıya) yığaq:

```
geom = [r1, r2, s1, s2]
```

for dövründə bu siyahının elementlərini ardıcıl sadalamaqla hər fiqurun perimetrinin qiymətini ekrana xaric etmək olar:

```
for g in geom:
    if isinstance(g, Rectangle):
        print(g.get_rect_pr())
    else:
        print(g.get_sq_pr())
```

Həmin nəticəni alacayıq.

Yeni sinif (Triangle) əlavə edək və onda üçbucağın perimetrini hesablamaq üçün get_tr_pr() metodunu təsvir edək:

```
class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c
    def get_tr_pr(self):
        return self.a + self.b + self.c
t1 = Triangle(1,2,3)
t2 = Triangle(4,5,6)
print(r1.get_tr_pr(), r2.get_tr_pr())
```

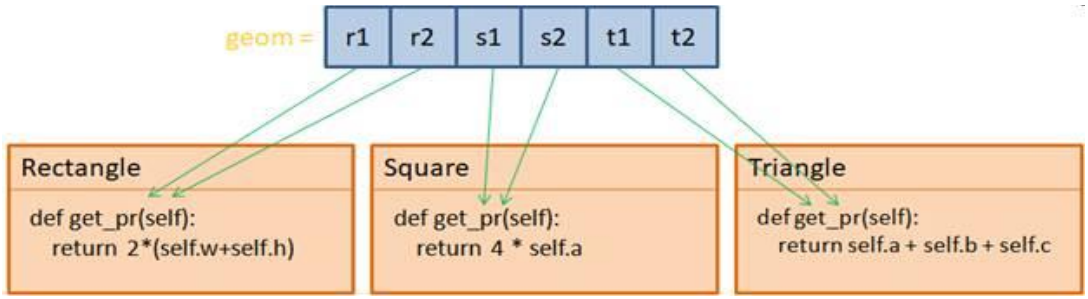
Nəticə:

```
6 15
```

İndi əgər t1 və t2-ni də geom siyahısına qatsaq və for dövründə əlavə olaraq Square və Triangle siniflərinə (uyğun olaraq) aid olduğunu yoxlamaq olar, lakin bu, proqramımıza gözəllik və çeviklik əlavə etməyəcək və proqram kodu lap uzanacaq. Bu halda polimorfizm adlanan yanaşmanın tətbiqi əvvərişli olur. Hər yaratdığımız siniflərdə eyni adlı (məsələn, get_pr()) metodlar yaradaq. Onda sadəcə olaraq for dövründə bu metoda müraciət edəcəyik (onu çağıracayıq) və uyğun fiqurların perimetrlərini alacayıq:

```
for g in geom:
print( g. get_pr() )
```

DOI:10.30546/gdu.2023.12-20



Yuxarıdakı şəkildə göründüyü kimi siyahının hər bir istinadı (keçidi) müvafiq sinif obyektinə aparır və sonra onun vasitəsilə get_pr() metoduna birbaşa çağırış baş verir. Bu polimorfizm nümunəsidir, biz vahid **geom** siyahısının indeksi (vahid interfeys) vasitəsilə müxtəlif obyektlərə daxil oluruq və sonra müvafiq obyektin **get_pr()** metodunu çağırırıq. Bundan əlavə, elə siyahıda müvafiq siniflərin obyektlərini yaratmaqla (yəni siyahıda əlavə dəyişənlərdən istifadə etməməklə) bu siyahını yarada bilərik:

geom = [Rectangle(1, 2), Rectangle(3, 4), Square(10), Square(20), Triangle(1, 2, 3), Triangle(4, 5, 6)]

Alınan nəticə:

6
14
40
80
15

Ədəbiyyat

1. K.Tahiroğlu “Python ilə proqramlaşdırma”, Bakı-2016.
2. Д. Бизли “Python. Подробный справочник”.4-ое издание, СПб Символ-Плюс, 2017.
3. Д. Ю. Федоров “Основы программирования на примере языка PYTHON”. .Учебное пособие, Санкт-Петербург, 2019.
4. A Byte of Python (Russian) Версия 2.02 Swaroop C.H. (Перевод: B. Смоляр), 2020.
5. Эрик Мэтиз «Изучаем Python», «Питер», Санкт-Петербург-2017.
6. Задорожный С.С., Фадеев Е. П. “Объектно-ориентированное программирование на языке Python” (учебно-методическое пособие), Москва, МГУ, 2022.
7. https://ru.wikipedia.org/wiki/Python
8. https://pythonru.com/primery/primery-raboty-s-klassami-v-python
9. https://docs-python.ru/tutorial/klassy-jazyke-python/
10. https://smartiqa.ru/courses/python/lesson-6
11. https://younglinux.info/oopython/oop
12. https://bestprogrammer.ru/izuchenie/chto-takoe-polimorfizm-v-python
13. https://younglinux.info/oopython/oop

НАСЛЕДОВАНИЕ И ПОЛИМОРФИЗМ В PYTHON

Бабаева Шакяр Маарифовна

Резюме

Под наследованием понимается возможность создания нового класса на базе существующего. При этом класс потомок будет содержать те же атрибуты и методы,

DOI:10.30546/gdu.2023.12-20

что и базовый класс, но при этом его можно (и нужно) расширять через добавление новых методов и атрибутов.

Полиморфизм – важный элемент объектно-ориентированного программирования в Python. Полиморфизм делает Python более универсальным и эффективным языком. Он позволяет использовать один оператор или функцию для выполнения нескольких задач. Это также позволяет вам повторно использовать имена методов, переопределяя их для уникальных целей разных классов.

Настоящая статья посвящена наследованию и полиморфизму в Python и созданию программ с использованием этих принципов ООР. В статье поясняется, что подходы, называемые наследованием и полиморфизмом, удобно применять к задачам, требующим наличия нескольких подобных групп объектов.

INHERITANCE AND POLYMORPHISM IN PYTHON

Babayeva Shakar Maarif

Summary

Inheritance refers to the ability to create a new class based on an existing one. In this case, the descendant class will contain the same attributes and methods as the base class, but it can (and should) be extended by adding new methods and attributes. Polymorphism is an important element of object-oriented programming in Python. Polymorphism makes Python a more versatile and efficient language. It allows you to use a single statement or function to perform multiple tasks. It also allows you to reuse method names by redefining them for the unique purposes of different classes. This article focuses on the principles of inheritance and polymorphism in Python and building programs using these principles. In the article, it is explained that it is convenient to apply approaches called inheritance and polymorphism on issues that require the presence of several similar groups of objects.

Rəyçi: Aslanov Ə.Ə.
İnformatika və cəbr kafedrasının 19 May 2023-cü il tarixli iclasın 09 saylı protok.
Daxil olma tarixi: 26 may 2023-cü il