Apstrakcija

Apstrakcija

Još jedan ključni koncept OOP-a jeste **apstrakcija**. Pod apstrakcijom podrazumevamo proces skrivanja odgovarajućeg dela koda od okruženja(drugih klasa i programa). Pre svega služi za pojašnjenje stvari na što jednostavniji način. Apstrakcija služi za postavljanje samih koncepata koji bi se tek kasnije implementirali.

Pri kodiranju postavlja se pitanje, da li mi u svakom trenutku moramo da znamo implementaiju nekog dela programa?

Uzmimo sledeći primer. Svi znamo da Python liste imaju ugrađenu metodu **.sort()**.

Da li je bitno da znamo njenu implementaciju u svakom slučaju? Dovoljno je da imamo neki opis šta nam metoda radi i koliko efikasno radi.

Apstrakcija u Python-u izvodi se pomoću takozvanih **ABC klasa** i **apstraktnih metoda**.

Apstraktne metode

Metode definisane sa ključnom reči **pass** odnosno definisane bez implementacije same akcije metode nazivaju se apstraktne metode.

One služe da definišu da neka klasa mora da ima potpis te metode, ali da nije neophodno da se istog trenutka vrši implementacija. Može se definisati korišćenjem dekoratera.

Primer:
```
@abstractmethod
def metoda(self):
```

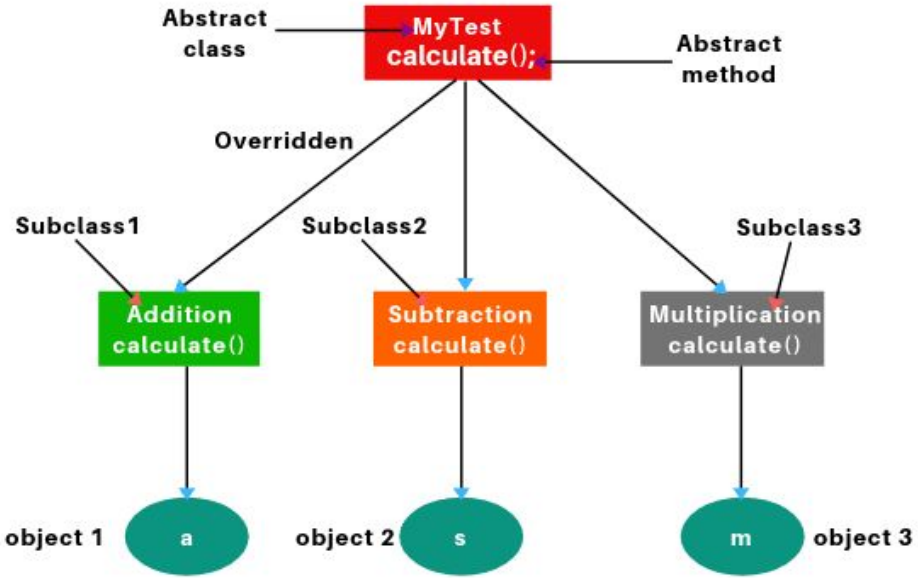

Fig: Abstract class and its subclasses

Apstraktne osnovne klase u Python-u (ABC)

Abstract Base Class (**ABC**) ili **apstraktne klase** u Python-u su klase u kome se postavljaju skice odnosno šabloni za definiciju nekog tipa. Za razliku od **konkretnih klasa**, metode apstraktnih klasa ne moraju da imaju nikakvu implementaciju(dovoljno je da se iskoristi **pass**).

Ideja iza apstraktnih klasa jeste da vršimo generalizaciju i da postavljamo šablon aktivnosti koji ne moramo istog trenutka da implementiramo.

Nekada prosto nije ni moguće za dati tip da implementiramo ponašanja istog trenutka, ali je bitno da naglasimo da ih klasa ima.

Navedimo primer da napravimo klasu **SahovskaFigura**. Možemo predvideti da šahovska figura mora imati metodu **pomeri_se()**. Međutim, pitanje je da li mi znamo kako objekat tipa **SahovskaFigura** treba da se pomera. Ono što znamo jeste kretanje konkretne figure poput kraljice, kralja ili pešaka.

Napomena: Apstraktna klasa ne sme imati neapstraktne metode, odnosno sve metode moraju imati ključnu reč **pass**.

Primer:
class **SahovskaFigura**(metaclass=**ABCmeta**):

Primer implementacije apstraktnih tipova

Navedeni kod prouzrokovaće **TypeError** grešku iz razloga što pokušavamo da instanciramo apstraktnu klasu.

Apstraktni tipovi(tipovi kojima se metaclass podesi na **ABCMeta**) ne mogu biti instancirani odnosno od takvih tipova se ne mogu praviti objekti.

```
from abc import ABCMeta, abstractmethod

class Zivotinja(metaclass=ABCMeta):

      @abstractmethod
      def odazovi_se(self):
            print("Ja sam zivotinja.")


a = Zivotinja()
a.odazovi_se()
```

Primer 9.

Napisati program u kome se definiše apstraktna klasa **SahovskaFigura**. U okviru klase deklarisati apstraktnu metodu **pomeri_se()** za koju treba da zna svaka šahovska figura i definisati . Zatim definisati klasu **Pesak** koja je **SahovskaFigura**, tako da se svaki pešak pomera na sebi odgovarajući način. Klasa ima parametrizovani konstruktor koji prihvata dva celobrojna argumenta koji predstavljaju poziciju figure na tabli. Napraviti 2 konkretna pešaka i zatim pomeriti jednog od njih.

Enkapsulacija

Enkapsulacija

Do sada je delimično obrađen pojam modularnosti. Konceptom **enkapsulacije** možemo da nadogradimo taj pojam i da svaki “modul” bude još manje zavisan .

Pod enkapsulacijom podrazumevamo skrivanje podataka i akcija u okviru jednog modula (tipa).

Za postizanje potpune enkapsulacije, treba obezbediti da odgovarajuće metode i promenljive ne budu pristupačne (“vidljive”) van definicije klase.

Modifikatori pristupa (vidljivosti)

Za implementaciju enkapsulacije, neophodno je da stvorimo doživljaj o pristupačnosti podataka, odnosno o njihovoj **vidljivosti**.

Uzećemo sledeću situaciju u obzir: postoji tip Osoba u okviru koga je definisana jedinstvena šifra koja predstavlja šifru za otključavanja ličnog telefona. Ukoliko bismo promenljivu sifra definisali kao što smo do sada, imali bismo glavni nedostatak da može bilo ko da pita osobu i da zatraži šifru, ujedno bismo mogli i da je direktno promenimo.

Sam koncept da su neki atributi ili metode nekima pristupačni, a nekima ne predstavlja **vidljivost** atributa odnosno metoda.

Naziv atributa/metode	Vidljivost	Ponašanje
naziv	javna	Moguć pristup i van klase i unutar klase
_naziv	zaštićena	Moguć, ali nepoželjan pristup direktno van klase
__atribut	privatna	Moguć pristup jedino unutar klase

Primer 10.

Napisati program u kome se definiše klasa **Telefon**. Svaki telefon ima privatnu promenljivu **PIN** koja je setovana na “0000”. Klasa **Telefon** zna za metodu **promeni_pin()**, koja prihvata dva argumenta među kojima je prvi argument string promenljiva koja predstavlja vrednost starog PIN-a, a drugi argument vrednost novog PIN-a. Ukoliko se vrednost starog PIN-a ne podudara sa samim PIN-om, ispisati poruku i onemogućiti menjanje šifre.

```
class Telefon:
      def __init__(self):
            self.__PIN = "0000"

      def promeni_pin(self, stari, novi):
            if stari == self.__PIN:
                  self.__PIN = novi
                  print("Uspesno promenjen PIN.")
            else:
                  print("Stari PIN je pogresno unet.")

t = Telefon()
t.promeni_pin("0000","1234")
t.promeni_pin("0000", "1233")
t.promeni_pin("1233", "2151")
```

Skrivanje metoda

Kao što skrivamo atribute u okviru jedne klase, imamo mogućnost da skrivamo i same metode.

Skrivanje metoda vršimo kada je potrebno da imamo neka ponašanja samo u okviru date klase. Dakle podrazumevamo da su to metode koje se više puta pozivaju u okviru jedne klase, ali ne želimo da drugi tipovi van klase mogu da pozovu tu metodu.

Primer:

```
def __moja_metoda(self, a):
        self.a = a + 10
```

Metode zahvata (GET) i postavljanja (SET)

Za sada smo naučili da sakrijemo atribute i metode van klase. Međutim, da li je to zapravo dovoljno za enkapsulaciju?

Ako u potpunosti sakrijemo podatke od sredine, mi onda nemamo nikakvu mogućnost da znamo šta su ti podaci. U takvim situacijama pravimo takozvane **metode zahvata**(**getter metode**). Analogno, ako hoćemo da menjamo vrednost nekog podatka pod nekim uslovom onda koristimo **metode postavljanja**(**setter metode**).

Primer: Pretpostavimo da objekat neke klase poseduje privatnu promenjivu vrednost. Getter i setter metode će redom izgledati:

```
def get_vrednost(self):
        return self.__vrednost

def set_vrednost(self, nova_vrednost):
        self.__vrednost = nova_vrednost
```

Primer X.

Napisati program u kome se definiše klasa **Osoba**. Svaka osoba ima privatne promenljive **ime_prezime** i **oznaka**. Takođe zna za privatno ponašanje **moja_oznaka_je()**, koje vraća string u formatu "*<oznaka>*". Dodatno definisati **getter** metodu za **ime_prezime** kao i **setter** metodu za **oznaku**. Override-ovati ToString metodu tako da vraća string u formatu "*ime_prezime <oznaka>*". Sama klasa ima parametrizovan konstruktor koji prihvata sve iznad spomenute promenljive.

Primer 11.

```
o1 = Osoba("Petar Petrovic", "P")
o2 = Osoba("Mira Maksimovic", "M")

print(o1.get_ime_prezime())
print(o1)

o2.set_oznaka("Z")
print(o2)
```

```
class Osoba:
	def __init__(self, ime, oznaka):
		self.__ime_prezime = ime
		self.__oznaka = oznaka

	def __moja_oznaka_je(self):
		return "<" + self.__oznaka + ">"

	def get_ime_prezime(self):
		return self.__ime_prezime

	def set_oznaka(self, oznaka):
		self.__oznaka = oznaka

	def __str__(self):
		return self.__ime_prezime + " " + self.__moja_oznaka_je()
```

Pregled bitnih pojmova i njihovi sinonimi 1

- Svojstva (sve promenljive i funkcije, …)
- Klasa (tip, skica, obrazac, …)
- Objekat (entitet, stvar, ...)
- Atribut (stanje, opis, ...)
- Metoda (ponašanje, akcija, radnja, …)
- Konstruktor (kreator, tvorac, …)
- Statička svojstva (klasni atributi i metode, ...)

Pregled bitnih pojmova i njihovi sinonimi 2

- Nasleđivanje (izvođenje, ...)
- Roditelj klasa (bazna klasa, natklasa ...)
- Dete klasa (izvedena klasa, potklasa, ...)
- Polimorfizam (višeobličnost, ...)
- Apstraktna klasa (nekonkretna klasa, ...)
- Apstraktna metoda (neimplementirana metoda, ...)
- Privatno svojstvo (skriveno, nedostupno van...)
- Javno svojstvo (neskriveno, dostupno van...)