



Distance Learning System

# Refleksija, događaji i dekoratori

Object Oriented Programming in Python

# Refleksija i analiza objekata

---

- Refleksija je sistem koji omogućava pristup klasama i objektima, i njihovu modifikaciju tokom izvršavanja programa
- Refleksija omogućava dinamičko instanciranje klasa i startovanje njihovih metoda
- Refleksija se smatra sporim sistemom u svim okruženjima u kojima postoji
- Refleksija nije naročito bezbedna, jer interveniše na strukturi klasa

# Analiza objekata

---

- Bitnije ugrađene funkcije za analizu objekata su: **dir, id, type, isinstance, issubclass, callable...**

# Funkcija id

- Vraća jedinstveni identifikator objekta tokom trajanja programa
- Važno je znati da je identifikator baziran najčešće na memorijskoj adresi, pa nije pogodan za perzistentnu upotrebu

```
class A:  
    x = 10  
    y = 20  
a = A()  
b = A()  
print(id(a))  
print(id(b))  
print(id(a.x))  
print(id(b.x))
```



```
4565778560  
4565778672  
4563366304  
4563366304
```

# Funkcija dir

- Dir vraća sve članove objekta prosleđenog kao parametar u obliku string liste

```
class A:  
    x = 10  
    y = 20  
  
a = A()  
  
print(dir(a))
```

→

```
['__class__', '__delattr__', '__dict__', '__dir__',  
 '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattr__', '__gt__', '__hash__', '__init__',  
 '__init_subclass__', '__le__', '__lt__', '__module__',  
 '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
 '__repr__', '__setattr__', '__sizeof__', '__str__',  
 '__subclasshook__', '__weakref__', 'x', 'y']
```

# Funkcija dir

- Izlistavanje polja je čest slučaj kod serijalizacije ili objektno relacionog mapiranja

```
output = ""
for i in dir(a):
    if not i.startswith("__"):
        val = getattr(a,i)
        output += f"{i}:{val},"
output = output.rstrip(",")
```

→ x:10,y:20


```
sql = "insert into points ({0}) values ({1})"
params = []
values = []
for i in dir(a):
    if not i.startswith("__"):
        val = getattr(a,i)
        params.append(f"{i}")
        values.append(f"'{val}'")
print(sql.format(",".join(params),"".join(values)))
```

→ insert into points (x,y) values ('10','20')

# Funkcija type

- Vraća tip objekta prosleđenog kao parametar

```
class A:  
    x = 2  
    y = 3  
a = A()  
  
t1 = type(A)  
t2 = type(a)  
  
print(t1)  
print(t2)
```



```
<class 'type'>  
<class '__main__.A'>
```

# Funkcija type

---

- Ukoliko ima sva tri parametra, funkcija generiše novi tip

```
calc = type('X', (), {"a":0,"b":0,"add":lambda self,a,b : a + b })  
c = calc()  
print(c.add(4,5))
```



# is funkcije

- Obzirom da je slabo tipiziran, Python ponekad zahteva rigorozne provere pre upotrebe nekog parametra
- U tu svrhu pomažu razne funkcije za analizu tipova
- Funkcija **isinstance**, proverava da li je neka vrednost određenog tipa

```
a = "10"
if isinstance(a,int):
    print(a*a)
else:
    print("Sorry bro, this is not a number")
```

- Funkcija **issubclass** proverava da li je neka klasa podklasa neke klase. Ova provera je korisna kako bi se osigurala tipizacija

# Funkcija callable

- Funkcijom callable, možemo da saznamo da li je neki objekat zapravo funkcija ili metoda:

```
class A:
    x = 10
    y = 20
    def f(self):
        print("Hello")

for i in dir(A):
    val = getattr(A,i)
    if not callable(val):
        print(f"{i} : ",end="")
        print(val)
```

# Modul inspect

<https://docs.python.org/3.8/library/inspect.html>

---

- Modul inspect proširuje mogućnosti analize objekata

```
import inspect
class A:
    x = 10
    def f(self):
        print(self.x)
a = A()
print(inspect.getmodule(a))
print(inspect.ismethod(a.f))
print(inspect.getfullargspec(a.f))
```

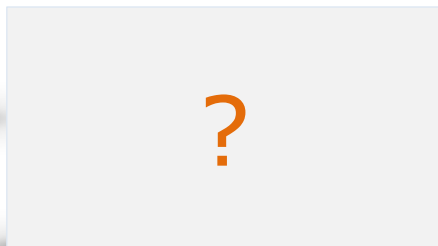
# Dinamičko učitavanje modula i instanciranje klasa

- Refleksija je korisna kada na primer želimo da instanciramo klasu i aktiviramo njen metod na osnovu evaluacije string parametara. Ovo je čest slučaj kada podaci dolaze spolja



→ ***BlackJack()***

→ ***Slot()***



→ *Nova klasa igre, ne može se instancirati pre nego što se predstavi sistemu*

# Instanciranje klase u refleksiji

(oopp-ex05 reflectioninstancing)

- Refleksija omogućava instanciranje klase i startovanje njenih metoda na osnovu stringova. Ovo omogućava da klasa za nas bude potpuno nepoznata, sve dok znamo parametre koji su nam potrebni za njeno instanciranje, i eventualno pozivanje njenih metoda

```
mod = "games"
game = "GameClass"
start_method = "start"
hit_method = "hit"
games_mod = __import__("games")
game_class = getattr(games_mod, game)
game_obj = game_class()
method = getattr(game_obj, start_method)
method()
method = getattr(game_obj, hit_method)
game_obj.hit(10)
game_obj.hit(20)
game_obj.hit(30)
```

# Vežba (oopp-ex05 lotterygames)

---

- Potrebno je kreirati dve jednostavne igre (nije neophodno kreirati logiku igara, već samo klase i prazne metode)
- 1: Gamble (Korisnik bira da li će sledeći broj biti veći ili manji od broja koji je odabrao računar)
- 2: Crvena crna (Korisnik bira da li će sledeća karta biti crvena ili crna)
- Obe igre imaju metod pick (u prvom slučaju, ovaj metod će uzeti input od korisnika i prikazati rezultat)
- Korisnik prilikom startovanja igre, unosi tip (klasu) igre koju hoće da igra (Gamble ili BlackRed)
- Ako igra nije prepoznata, program prikazuje grešku
- Za instanciranje igre treba koristiti refleksiju

# Događaji

- Događaji su pojam kome će biti posvećena pažnja u ovom, ali i u narednim kursevima. To je jedan od ključnih koncepata u programiranju GUI aplikacija
- Programiranje koje podrazumeva praćenje i obradu događaja u toku izvršenja aplikacije naziva se **Event Based** programiranje. Ovaj način rukovanja programom prepoznatljiv je u svim aplikacijama koje sadrže korisničke kontrole (tastere, prozore, tekst boksove...).
- Recimo da želimo da se vozimo automobilom kome rezervoar nije pun. Imali bismo dva načina da budemo sigurni da se nećemo zaustaviti. Jedan je da konstantno proveravamo stanje u rezervoaru, a drugi da se oslonimo na indikator rezerve goriva. Naravno, druga varijanta je daleko galantnija od prve i to je upravo način na koji funkcionišu i događaji objekata u programiranju.



# Callback funkcija

---

- Callback funkcija je funkcija koja se ne poziva direktno, već se prosleđuje kako bi bila pozvana od strane neke druge funkcije

```
def f(cbf):  
    cbf()
```

```
def my_cbf():  
    print("Hello from cbf")
```

```
f(my_cbf)
```

→ Hello from cbf



# Događaji unutar objekta

## (oopp-ex05 simplelocalevent.py)

- Analizirajmo sledeći program

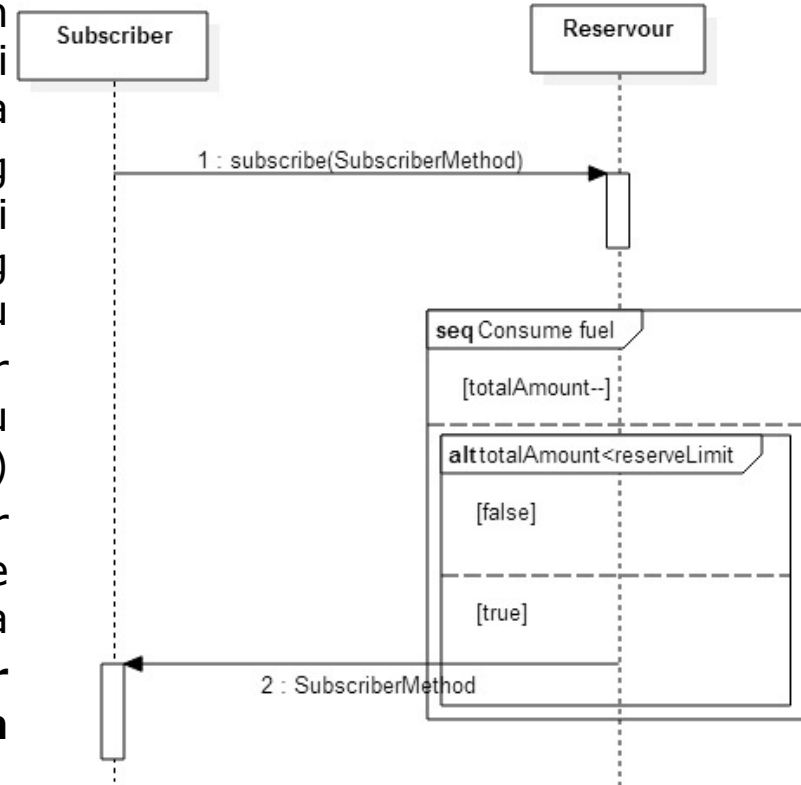
```
class Reservoir:
    def __init__(self):
        self.reserveLimit = 10
        self.totalAmount = 100
    def reserveIndicator(self):
        print("Hey, I am on reserve! Please refill me!")
    def getFuel(self):
        self.totalAmount -= 1
        if self.totalAmount <= self.reserveLimit:
            self.reserveIndicator()
        print(self.totalAmount)
```

```
res = Reservoir()
for i in range(100):
    res.getFuel()
```

- Primećujemo da je rezervoar svestan ulaska u rezervu, ali da svet oko njega nije. Unutar main metode, ne znamo da je rezervoar „na rezervi“ jer je događaj identifikovan i obrađen unutar objekta

# Distributer/subscriber model (oopp-ex05 carevents)

- Da bi događaj bio vidljiv za objekte izvan objekta u kome se dogodio (**distributera**), oni se moraju pretplatiti na njega
- Da bi se objekat pretplatio na događaj drugog objekta, mora ispuniti uslove. Ti uslovi podrazumevaju postojanje odgovarajućeg metoda na objektu pretplatniku
- U trenutku pretplate, objekat distributer prijavljuje pretplatnika na događaj (stavlja ga u listu pretplatnika)
- Prilikom detektovanja događaja, distributer prolazi kroz listu pretplatnika i svima im šalje informaciju da je došlo do događaja
- Ovaj koncept poznat je i pod nazivom **observer pattern**



# Slušać događaja

---

- Prvi korak u procesu definicije događaja je konstrukcija slušača događaja. Slušać događaja je najčešće apstrakcija koja ima jedan ili više metoda za koje će znati distributer i pretplatnici. Slušać događaja je nešto što treba da bude logički vezano za klasu koja će generisati događaj (ili više događaja)

```
class EventListener(abc.ABC):  
    @abc.abstractmethod  
    def reserve_reached(self, sender):  
        pass
```

# Distributer događaja

- Distributer je klasa u kojoj se događaj dogodio. Ova klasa mora imati **listu slušača**, mehanizam za pridruživanje slušača listi (**add\_listener**), mehanizam za uklanjanje slušača (**remove\_listener**) i mehanizam za obaveštavanje slušača.

```
def add_listener(self, listener):  
    self.listeners.append(listener)  
  
def remove_listener(self, listener):  
    self.listeners.remove(listener)  
  
def distribute_event(self):  
    for listener in self.listeners:  
        if isinstance(listener, EventListener):  
            listener.reserve_reached(self)
```

# Aktivacija događaja

- Kada postoji kompletan mehanizam za upravljanje događajima unutar klase, samu aktivaciju događaja treba vršiti po potrebi

```
class Reservoir:
    def __init__(self):
        self.current_state = 100
        self.reserve_limit = 50
        self.listeners = []

    def add_listener(self, listener):
        self.listeners.append(listener)

    def remove_listener(self, listener):
        self.listeners.remove(listener)

    def distribute_event(self):
        for listener in self.listeners:
            if isinstance(listener, EventListener):
                listener.reserve_reached(self)

    def consume_fuel(self):
        print(f"Fuel consumed. {self.current_state} liters remaining")
        self.current_state -= 1
        if self.current_state < self.reserve_limit:
            self.distribute_event()
```

# Pretplata na događaj

- Kada je objekat u stanju da detektuje događaj i distribuirati ga pretplatnicima, same pretplatnike možemo (a ne moramo) dodavati prema potrebi

```
res = reservoir.Reservoir()

class MyListener(reservoir.EventListener):
    def reserve_reached(self, sender):
        print("No more fuel in car. Please refill!")

res.add_listener(MyListener())

for i in range(0,100):
    res.consume_fuel()
    time.sleep(0.1);
```

# Vežba (oopp-ex05 hitme)

- Aplikacija traži od korisnika x poziciju tenka
- Nakon unosa, računar odabira poziciju svog tenka
- Pozicije oba tenka se zatim ispisuju na izlazu
- Korisnik unosi jačinu i ugao svog sledećeg hica
- Hitac se ispaljuje i prati u realnom vremenu, proveravajući za svaki pomeraj, da li je:
  - Projektil udario u tlo
    - Hitac se smatra završenim i ne dodeljuju se poeni za pogodak
  - Projektil udario u neprijateljski tenk
    - Hitac se smatra završenim i dodeljuju se poeni za pogodak
  - **Opciono:**
    - Sistem generiše random pozicije na „nebu“, koje predstavljaju ptice
    - Projektil udario u pticu
      - U ovom slučaju, broj poena za taj hitac, povećava se za svaku pticu po jedan
- Pomenute dve (tri) situacije rešiti pomoću event-a

# Dekoratori

---

- Dekoratori su meta podaci metoda
- Koriste se da bi dali neku informaciju sistemu prilikom izvršavanja, ali da pri tom ne utiču direktno na funkcionalnost i strukturu same klase





# Korišćenje dekoratora

---

- Dekorator možemo prepoznati po oznaci @ ispred naziva, i poziciji, koja je obično iznad klase ili njenog člana
- Na primer:

```
@staticmethod  
def my_method():  
    print("Hello")
```

**Dekorator**



# Korisnički definisani dekoratori

(oopp-ex05 taxdecorator.py)

- Možemo kreirati i sopstvene dekoratore
- Dekorator je wrapper funkcije

```
def tax(func,*args):  
    def wrapper(*args):  
        return func(args[0],args[1],args[2]) * 1.2  
    return wrapper  
  
class Calc(object):  
    @tax  
    def do(self, x, y):  
        return x + y  
  
c = Calc()  
print("{:.2f}".format(c.do(4,5)))
```

**Definicija  
dekoratora**

**Postavka  
dekoratora**

# Parametrizacija korisnički definisanog dekoratora

(oopp-ex05 pdecorator.py)

```
def decorator(oper):
    def inner(f):
        if oper == "+":
            return lambda a,b : a + b
        elif oper == "-":
            return lambda a,b : a - b
        return f
    return inner

@decorator("-")
def f1(a,b):
    pass

@decorator("+")
def f2(a,b):
    pass

print(f1(4,3))
print(f2(4,3))
```

- Dekoratore je moguće parametrizovati
  - Tada treba obratiti pažnju na telo dekoratora i ono što on vraća
1. Dekorator biva pozvan
  2. Dekorator startuje funkciju dobijenu kao rezultat, prosleđujući joj izvornu funkciju

# Dekorator objekat

(oopp-ex05 odecorator.py)

```
class decorator:
    def __init__(self,op):
        self.op = op

    def __call__(self,f):
        if self.op == "+":
            return lambda a,b : a + b
        elif self.op == "-":
            return lambda a,b : a - b

@decorator("-")
def f1(a,b):
    pass

@decorator("+")
def f2(a,b):
    pass

print(f1(4,3))
print(f2(4,3))
```

- Moguće je kompletnu klasu kreirati isključivo u svrhu dekoratora
- Tada se prilikom aktivacije dekoratora (poziva dekorisane funkcije / metode) instancira objekat

# Vežba

## (oopp-ex05 messageservice)

---

- Potrebno je kreirati dekorator pod nazivom **locale**
- Dekorator mora da ima jedan parametar **language**
- Potrebno je kreirati klasu **UserMessageService** koja sadrži poruke dobrodošlice na različitim jezicima pri čemu su ključevi oznake jezika (en,fr...), a vrednosti same poruke Hello, Bonjour...
- Klasa treba da ispisuje sadržaj na izlaz, pomoću metode **show\_welcome\_message**
- Klasa mora da ima anotaciju locale, u kojoj će biti kao parametar naveden jezik pozdravne poruke
- Klasu treba instancirati, a zatim prikazati pozdravnu poruku na jeziku određenom u anotaciji