Objektno-orijentisano programiranje i Python

© Goodrich, Tamassia, Goldwasser

Katedra za informatiku, Fakultet tehničkih nauka, Univerzitet u Novom Sadu

2018.

Terminologija

- svaki objekat koji se kreira u programu je instanca nečega što zovemo klasa
- **klasa** spoljašnjem svetu predstavlja pogled na objekte koji su njene instance
- bez nepotrebnih detalja ili davanja pristupa unutrašnjosti
- klasa sadrži atribute (instance variables, data members) i metode (member functions) koje objekat može da izvrši

Ciljevi

- robusnost
  - želimo da softer može da prihvati neočekivane ulazne podatke koji nisu ranije bili predviđeni
- adaptivnost
  - želimo da softer može da evoluira tokom vremena kao odgovor na promene u zahtevima ili okruženju
- ponovna iskoristivost (reusability)
  - želimo da omogućimo da se isti programski kôd koristi kao komponenta u različitim sistemima ili primenama

Apstraktni tipovi podataka

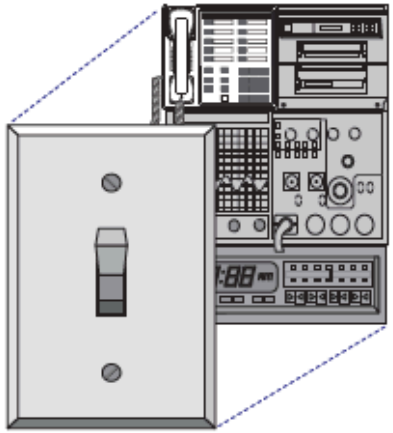
- apstrakcija predstavlja izdvajanje najvažnijih osobina nekog sistema
- primena apstrakcije na dizajn struktura podataka dovodi do apstraktnih tipova podataka (ATP)
- ATP je model strukture podataka koji definiše tip podataka, operacije nad njima, i tipove parametara tih operacija
- ATP definiše **šta** operacija radi, ali ne i **kako** to radi
- skup operacija koje definiše ATP je interfejs (public interface)

Principi objektno-orijentisanog dizajna

- modularnost
- apstrakcija
- enkapsulacija

Modularity

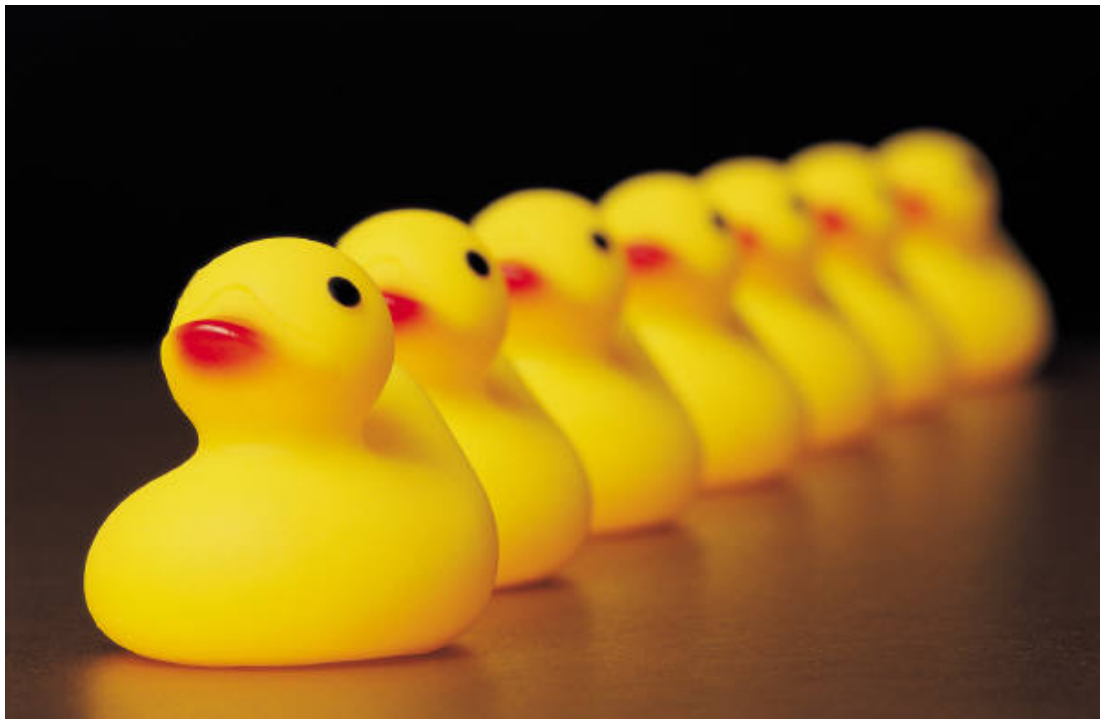
Abstraction



Encapsulation

Duck typing

- Python rukuje apstrakcijama pomoću duck typing principa
  - pesnik James Whitcomb Riley: "when I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck"



Duck typing

- program tretira objekte kao da imaju određenu funkcionalnost ako se ponašaju ispravno i ispunjavaju traženo
- Python je interpretirani jezik sa dinamičkim tipovima
  - nema compile-time provere tipova podataka
  - nema posebnih formalnih zahteva kod definisanja novih tipova podataka

Apstraktne bazne klase

- Python radi sa ATP pomoću mehanizma apstraktnih baznih klasa (abstract base classes, ABC)
- ABC se ne može instancirati, ali definiše zajedničke metode koje sve implementacije te apstrakcije moraju imati
- ABC se realizuje pomoću jedne ili više konkretnih klasa koje nasleđuju ABC i implementiraju metode koje propisuje ABC
- možemo koristiti postojeće ABC i postojeće konkretne klase iz Python-ove biblioteke

Enkapsulacija

- komponente softverskog sistema ne bi trebalo da otkrivaju detalje svog unutrašnjeg funkcionisanja
- neki aspekti strukture podataka su javni
- a neki predstavljaju interne detalje i privatni su
- Python delimično podržava enkapsulaciju
  - konvencija: atributi i metode koje počinju donjom crtom (npr. `_secret`) su privatni i ne treba ih koristiti izvan klase

Šabloni dizajna (design patterns)

algoritamski šabloni	šabloni dizajna
rekurzija	iterator
amortizacija	adapter
podeli pa vladaj	position
odseci pa traži	composition
gruba sila	template method
dinamičko programiranje	locator
pohlepni metodi	factory method

Objektno-orijentisani dizajn softvera

- odgovornost: podeliti posao različitim učesnicima, svako sa različitim odgovornostima
- nezavisnost: definisati namenu svake klase što je moguće više nezavisno od drugih klasa
- ponašanje: definisati ponašanje svake klase pažljivo i precizno, tako da posledice svake akcije budu dobro shvaćene od strane drugih klasa

Objedinjeni jezik za modelovanje (UML)

- Unified Modeling Language (UML): (grafički) jezik za opis softverskih sistema
- prikaz klase na UML dijagramu ima tri celine:
  - ime klase
  - atribute
  - metode

Class:	CreditCard	
Fields:	_customer _bank _account	_balance _limit
Behaviors:	get_customer( ) get_bank( ) get_account( ) make_payment(amount)	get_balance( ) get_limit( ) charge(price)

Definicije klasa

- klasa je osnovno sredstvo apstrakcije u OOP
- u Pythonu je svaki podatak predstavljen instancom neke klase
- klasa definiše ponašanje pomoću **metoda**; sve instance imaju iste metode
- klasa definiše stanje pomoću **atributa**; svaka instanca ima svoju kopiju atributa

Identifikator `self`

- svaka klasa može imati više svojih instanci
- svaka instanca ima svoj primerak atributa
- stanje svake instance predstavljeno je vrednošću njenih atributa
- `self` predstavlja instancu za koju je metoda pozvana

Primer klase 1

```
class CreditCard:
  """Predstavlja bankarsku kreditnu karticu."""

  def __init__(self, customer, bank, acnt, limit):
    """Kreira novu instancu kartice.

    Početno stanje na računu je nula.

    customer  ime klijenta ('Žika Žikić')
    bank      ime banke ('ABC Banka')
    acnt      broj računa ('5931 0375 9837 5309')
    limit     ograničenje kredita
    """
    self._customer = customer
    self._bank = bank
    self._account = acnt
    self._limit = limit
    self._balance = 0
```

Primer klase 2

```
def get_customer(self):
  """Vraća ime klijenta."""
  return self._customer

def get_bank(self):
  """Vraća ime banke."""
  return self._bank

def get_account(self):
  """Vraća broj računa."""
  return self._account

def get_limit(self):
  """Vraća ograničenje kredita."""
  return self._limit

def get_balance(self):
  """Vraća stanje računa."""
  return self._balance
```

Primer klase 3

```
def charge(self, price):
  """Naplati datu cenu na kartice uz poštovanje limita.

  Vraća True ako je novac naplaćen; False ako nije.
  """
  if price + self._balance > self._limit:
    return False
  else:
    self._balance -= price
    return True

def receive(self, amount):
  """Uplaćuje novac u datom iznosu na račun."""
  return self._balance += amount
```

Konstruktori

- kreiranje instanci klase `CreditCard`:

```
cc = CreditCard('Žika Žikić',
                'ABC Banka',
                '5931 0375 9837 5309',
                1000)
```

- interno će se ovo prevesti na poziv metode `__init__`
- njen zadatak je da novokreirani objekat dovede u korektno početno stanje postavljanjem odgovarajućih vrednosti atributa

Preklapanje operatora

- eng. operator overloading
- ugrađene Python klase imaju definisane operatore sa prirodnom semantikom
- na primer, izraz `a + b` predstavlja sabiranje kod brojčanih podataka, a konkatenaciju kod stringova i lista
- kada pišemo svoju klasu možemo da definišemo operator + za instance naše klase

Iteratori

- iterator za bilo kakvu kolekciju podataka omogućava da se svaki element kolekcije dobije tačno jednom
- potrebno je napisati metodu `__next__` koja vraća sledeći element kolekcije
- ili izaziva izuzetak `StopIteration` ako nema više elemenata

- umesto `__next__` mogu se napraviti `__len__` i `__getitem__`

Iteratori: primer

```
class Range:
"""Klasa koja oponaša ugrađenu Range klasu."""

  def __init__(self, start, stop=None, step=1):
    """Inicijalizuje Range instancu."""
    if step == 0:
      raise ValueError('step cannot be 0')
    if stop is None:
      start, stop = 0, start # range(n) isto što i range(0, n)
    self._length = max(0, (stop-start+step-1)//step) # zapamti dužinu
    self._start = start # treba da zapamtimo start i step zbog __getitem__
    self._step = step

  def __len__(self):
    """Vraća broj elemenata."""
    return self._length

  def __getitem__(self, k):
    """Vraća element na poziciji k."""
    if k < 0: # za negativan k broji se od nazad
      k += len(self)
    if not 0 <= k < self.length:
      raise IndexError('index out of range')
    return self._start + k + self._step
```

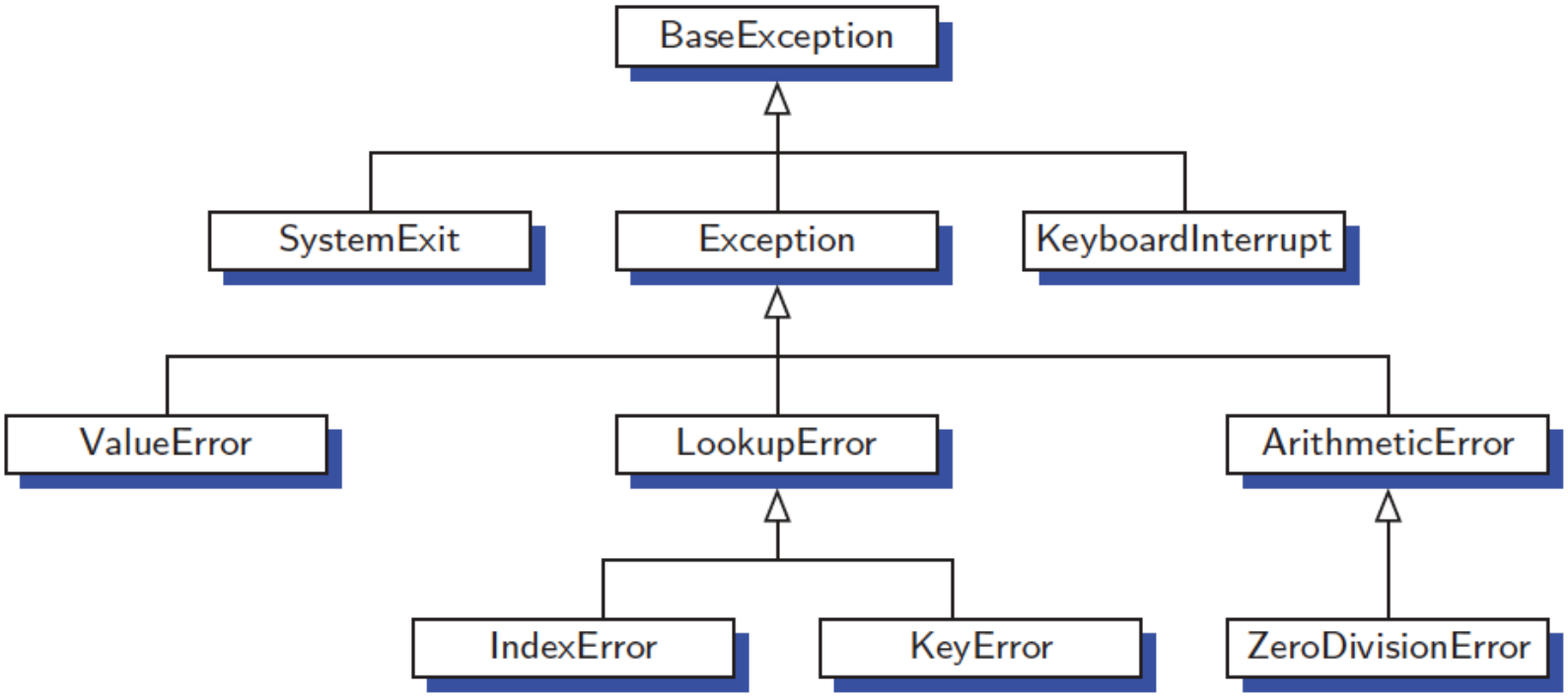
Nasleđivanje

- nasleđivanje je mehanizam za modularnu i hijerarhijsku organizaciju
- omogućava da se nova klasa definiše pomoću postojeće kao početne tačke
- postojeća klasa se obično zove **bazna**, **roditeljska** ili `superklasa`
- nova klasa se obično zove **potklasa**, **dete**-klasa ili **naslednik**

- postoji dva načina da se potklasa učini različitom od roditelja
  - potklasa može da promeni ponašanje tako što će imati novu implementaciju neke nasleđene (postojeće) metode
  - potklasa može da proširi roditelja dodavanjem novih metoda ili atributa

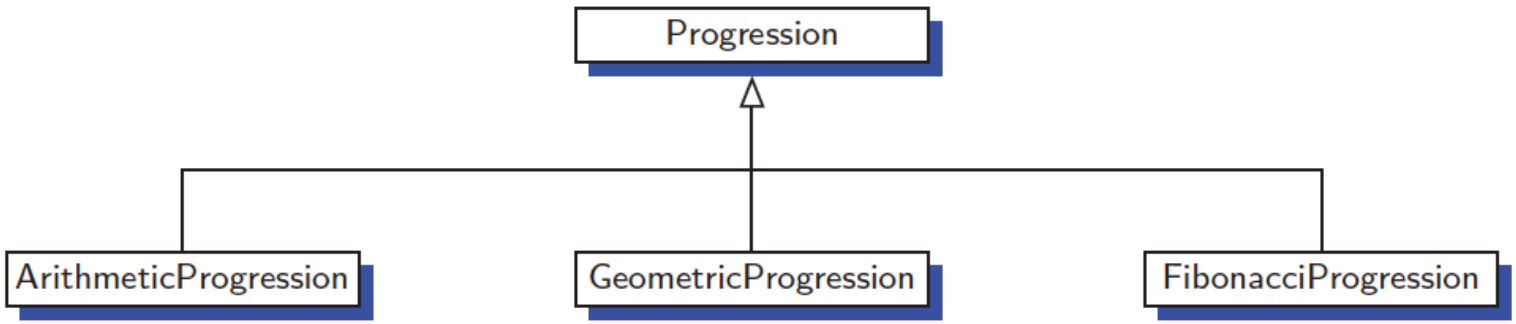
Python već koristi nasleđivanje

- hijerarhija klasa koje predstavljaju izuzetke



Primer 2

- numerička progresija je niz brojeva kod koga vrednost svakog elementa zavisi od vrednosti jednog ili više prethodnih elemenata
- **aritmetička** progresija određuje sledeći broj dodavanjem fiksne konstante na prethodni broj
- **geometrijska** progresija određuje sledeći broj množenjem prethodnog broja fiksnom konstantom
- **Fibonačijeva** progresija koristi formulu $F_{i+1} = F_i + F_{i-1}$



Primer 2: bazna klasa

```
class Progression:
"""Iterator koji predstavlja generičku progresiju.
   Po defaultu proizvodi niz brojeva 0, 1, 2, ...
"""

  def __init__(self, start=0):
    """Inicijalizuje tekući broj na prvi broj progresije."""
    self._current = start

  def _advance(self):
    """Izračunava novi tekući broj self._current.
    Ovo treba da redefiniše klasa naslednica.
    Po konvenciji, None označava kraj progresije.
    """
    self._current += 1

  def __next__(self):
    """Vraća sledeći element ili izaziva StopIsolation izuzetak."""
    if self._current is None:
      raise StopIteration()
    else:
      answer = self._current
      self._advance()
      return answer

    def __iter__(self):
      """Po konvenciji iterator mora da vrati sebe kao iteratora."""
      return self

    def print_progression(self, n):
      """Ispisuje sledećih n vrednosti u progresiji."""
      print(' '.join(str(next(self)) for j in range(n)))
```

Primer 2: potklasa za aritmetičku progresiju

```
class ArithmeticProgression(Progression): # nasleđuje Progression
"""Iterator koji proizvodi aritmetičku progresiju."""

  def __init__(self, increment=1, start=0):
    """Kreira novu aritmetičku progresiju.
       increment  fiksna konstanta koja se sabira (default je 1)
       start      prvi element progresije (default je 0)
    """
    super().__init__(start) # poziv konstruktora bazne klase
    self._increment = increment

  def _advance(self): # redefinišemo nasleđenu metodu
    """Izračunava novi tekući broj dodajući increment. """
    self._current += self._increment
```

Primer 2: potklasa za geometrijsku progresiju

```
class GeometricProgression(Progression): # nasleđuje Progression
"""Iterator koji proizvodi geometrijsku progresiju."""

  def __init__(self, base=2, start=1):
    """Kreira novu geometrijsku progresiju.
       base       fiksna konstanta kojom se množi (default je 2)
       start      prvi element progresije (default je 1)
    """
    super().__init__(start) # poziv konstruktora bazne klase
    self._base = base

  def _advance(self): # redefinišemo nasleđenu metodu
    """Izračunava novi tekući broj množeći ga sa base. """
    self._current *= self._base
```

Primer 2: potklasa za Fibonačijevu progresiju

```
class FibonacciProgression(Progression): # nasleđuje Progression
"""Iterator koji proizvodi Fibonačijevu progresiju."""

  def __init__(self, first=0, second=1):
    """Kreira novu Fibonačijevu progresiju.
       first      prvi element progresije (default je 0)
       second     drugi element progresije (default je 1)
    """
    super().__init__(first)
    self._prev = second-first # izmišljena vrednost pre prve

  def _advance(self): # redefinišemo nasleđenu metodu
    """Izračunava novi tekući broj sabirajući prethodna dva. """
    self._prev, self._current = self._current, self._prev + self._current
```