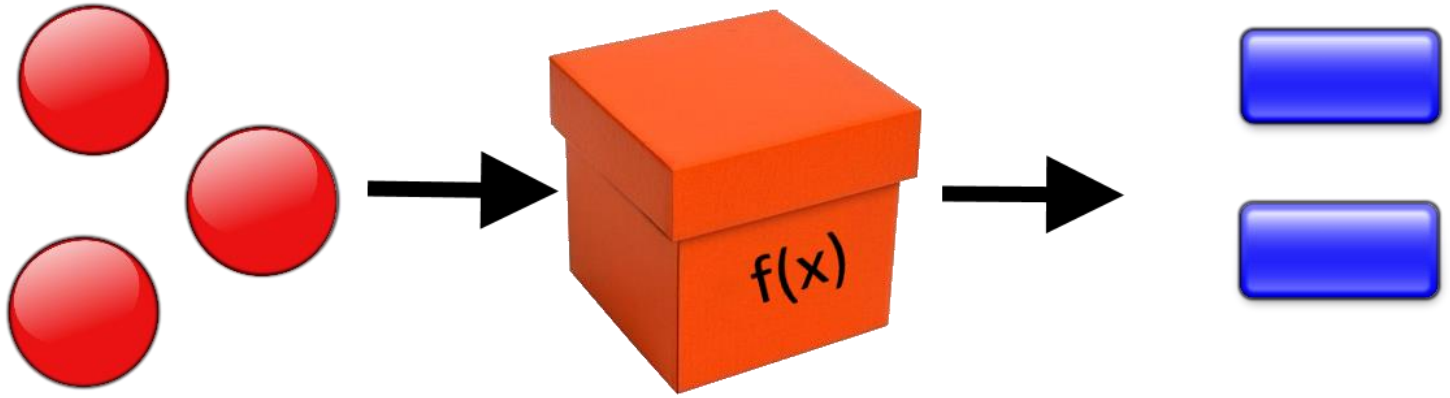
Katedra za računarstvo

Elektronski fakultet, Univerzitet u Nišu

Veštačka inteligencija

Python – Funkcionalno programiranje

Funkcije



- Python nije isključivo funkcionalni jezik, ali ima veliki broj funkcionalnih karakteristika
- Za razliku od imperativnog programiranja, koje se bazira na izvršavanju niza naredbi, funkcionalno programiranje vrši evaluaciju funkcijskih izraza
- Nekorišćenje globalnih promenjivih eliminiše kompleksnost i onemogućava probleme koji mogu da se jave prilikom pristupanja
- To omogućava jednostavnu paralelizaciju koda, bez sporednih efekata koji mogu da se tom prilikom jave

Funkcije

- Korišćenje rekurzije omogućava podelu problema na sitnije, lakše rešive verzije istog problema
- Poreklo vodi iz matematike
- Postoji više vrsta rekurzije
  - Primitivna (rekurzija glave, head recursion) se jednostavno koristi kod transformacije petlji u rekurzivne pozive
    - Rekurzivni poziv je prvi poziv u funkciji, čiji rezultat se koristi u daljoj evaluaciji
  - Repna rekurzija (tail recursion) – rekurzivni poziv je poslednji poziv u funkciji

Funkcije

- Jednostruka rekurzija – postoji jedan rekurzivni poziv u funkciji
- Višestruka rekurzija – postoji više rekurzivnih poziva funkcije
- Indirektna rekurzija – rekurzija kod koje se 2 ili više funkcija naizmenično pozivaju jedna iz druge

- Funkcionalni jezici omogućavaju i veliki broj optimizacija. Repna rekurzija je jedna od njih
  - Nažalost, Python ne podržava automatsku optimizaciju repne rekurzije, pa je neophodno kod prevesti u taj oblik ručno

Repna rekurzija – tail recursion

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return factorial(n - 1) * n

# Verzija sa repnom rekurzijom (bez optimizacije)
def factorialRR(n, acc = 1):
  if n == 0:
    return acc
  else:
    return factorialRR(n - 1, acc * n)
```

Funkcije prve klase

- Funkcije u Python programskom jeziku su funkcije prve klase
- Python podržava:
  - Slanje funkcija kao argumenata u druge funkcije
  - Vraćanje funkcije kao rezultata druge funkcije
  - Dodeljivanje funkcija promenjivama, menjanje imena funkcija, kopiranja funkcija u druge promenjive

```
>>> print.__qualname__
'print'
>>> stampaj = print
>>> stampaj.__qualname__
'print'
```

Čiste funkcije

- Čiste funkcije ne proizvode posledice (ne menjaju stanja izvan tela funkcije)
- Komunikacija sa ostatkom programa se vrši isključivo putem argumenata, koji takođe ne smeju da se menjaju, kao i povratne vrednosti, kojom rezultat prosleđuju nazad

- Korišćenje čistih funkcija omogućava bezbedno izvršavanje konkurentnog koda kod paralelnog programiranja

Funkcije višeg reda

- Funkcije koje prihvataju druge funkcije kao argument ili vraćaju funkciju kao rezultat
- Funkcije višeg reda postoje zahvaljujući konceptu funkcija prve klase
- Postojanje funkcija prve klase zahteva postojanje funkcija višeg reda, ali obrnuto ne važi

Monad

- Postojanje čistih funkcija je jako bitno za funkcionalno programiranje, zbog eliminacije posledica koje mogu da nastanu
- Nažalost, lako je dodati funkcionalnost koja će da uvede posledice u čistu funkciju

```
def kvadrat(x):
    return x ** 2

def kvadratP(x):
    print ("Broj za kvadriranje je " + str(x)) # I/O posledica
    return x ** 2
```

Monad

- Da bi se zadržala čista funkcija, ona mora da sve ulazne podatke dobija preko argumenata i da jedini rezultat bude povratni podatak
- Omogućićemo da funkcija može da vrati više od kvadriranog broja, string koji će da sadrži broj koji je prosleđen

```
def kvadratPP(x: int) -> (int, str):      # Anotacije (:int i -> (int, str))
    broj = "Broj: " + str(x)              # kvadratPP.__annotations__
    return (x ** 2, broj)                 # za više informacija o metodi
```

- Na ovaj način smo eliminisali problem koji je nastao uvođenjem print naredbe, ali i onemogućili ulančavanje funkcija zato što se tip koji funkcija vraća razlikuje od tipa argumenta

Monad

- Rešenje je funkcija koja prihvata obe funkcije i kombinuje ih u jednu

```
def kombinacijaFunkcija(drugaFja, prvaFja, broj: int):
    prviRez = prvaFja(broj)
    drugiRez = drugaFja(prviRez[0])
    return (drugiRez[0], prviRez[1] + ", " + drugiRez[1])

>>> kombinacijaFunkcija(kvadratPP, kvadratPP, 2)
(16, 'Broj: 2, Broj: 4')
```

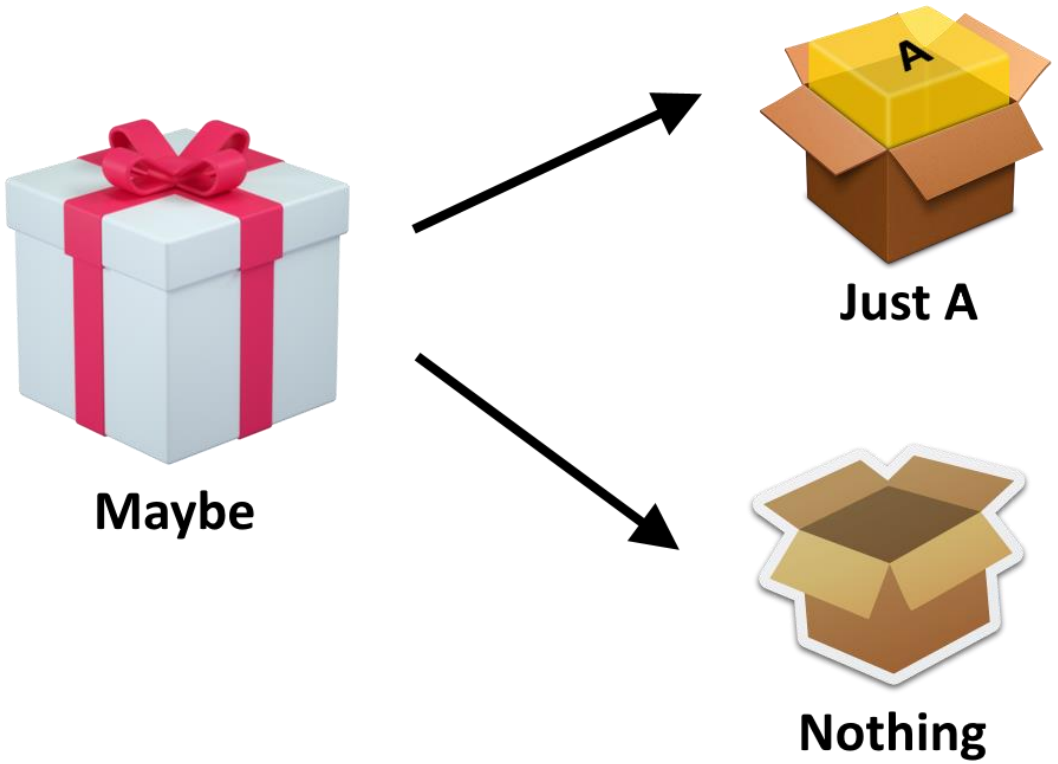
- Na ovaj način smo dobili rezultat koji je kombinacija 2 ulančana poziva funkcije kvadratPP, ali šta ukoliko nam je potrebno više?

Monad

- Rešenje su 2 funkcije:
  - Funkcija unit, koja broj koji se prosleđuje na početku transformiše u tuple podatak i dodaje mu string koji će da pamti brojeve,
  - Funkcija bind koja će da prihvati funkciju koju treba izvršiti i podatak u takvom obliku (bez obzira da li ga je kreirao unit ili neki drugi poziv bind funkcije).

- Bind izvršava funkciju nad prvim podatkom iz tuple (broj) i vraća novi tuple koji se sastoji iz rezultata i prethodnog string-a na koji nadovezuje novi broj
- Ovime se dobija mogućnost ulančavanja neograničenog broja funkcija bez posledica po čiste funkcije

```
def kvadratPP(x: int) -> (int, str):
    broj = "Broj: " + str(x)
    return (x ** 2, broj)

def bind(fja, prethodni: (int, str)):
   rez = fja(prethodni[0])
   return (rez[0], prethodni[1] +
            (", " if prethodni[1] else "") +
              rez[1])

def unit(broj: int):
   return (broj, "")

>>> print(bind(kvadratPP,
         (bind(kvadratPP,
         (bind(kvadratPP,
               unit(2)))))))
(256, 'Broj: 2, Broj: 4, Broj: 16')
```



Prednosti i mane funkcionalnog programiranja

Prednosti

+ Apstrakcija – rad sa kolekcijama podataka se svodi na poziv funkcije, manje koda, manje šanse za greškama
+ Jednostavno debagiranje, prepravke koda
+ Jednostavna paralelizacija

Mane

- Input/Output sistem ne može da se implementira na funkcionalan način
- Rekurzija je uglavnom sporija od korišćenja iteracija
- Stanje je teško zapamtiti u funkcionalnom programiranju. Ono se prenosi sa izlaza jedne na ulaz druge funkcije

Druge optimizacije

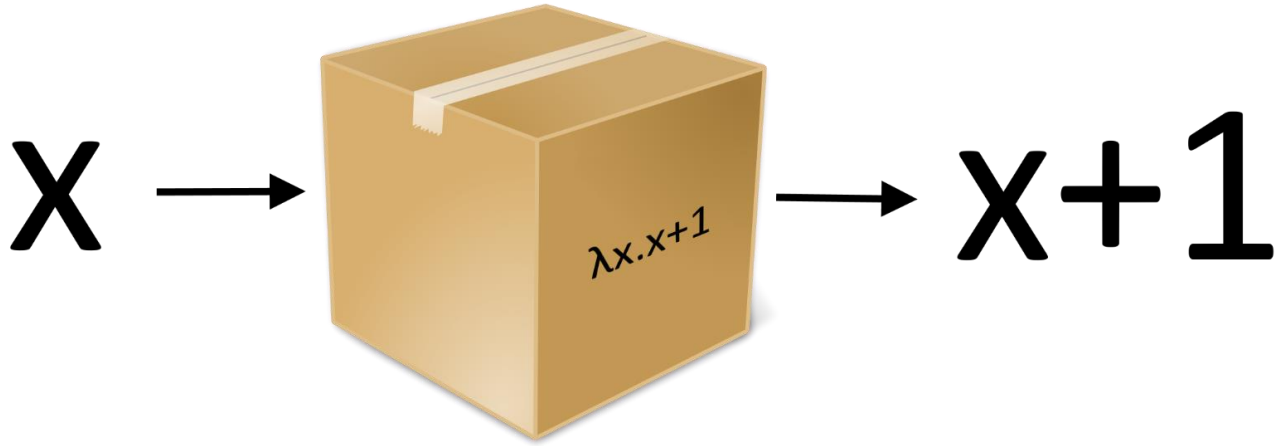
- Memoizacija
  - Svako ponovljeno izvršavanje funkcije se zamenjuje rezultatom koji je dobijen ranije i sačuvan
  - Može da dovede do drastičnih poboljšanja performansi kod nekih programa
- Lenjo izračunavanje
  - Odlaganje izračunavanja izraza, sve dok njegova vrednost nije neophodna za dalje izvršenje programa. Ukoliko nema potrebe za rezultatom, neće nikada biti izvršen
  - Omogućava dalje optimizacije, kombinacijom više izraza u jedan jednostavniji pre izračunavanja, kod nekih izraza
  - Primer: `range(1, 100)` - Ni jedan element nije kreiran, do trenutka kada je potreban

Memoizacija

```
def fibonacciMemo(n, memo = {}):
    if (n <= 1):
        memo[n] = n
        return n
    else:
        if ((n - 1) in memo.keys()):
            nm1 = memo[n - 1]
        else:
            nm1 = fibonacciMemo(n - 1, memo)
            memo[n - 1] = nm1
        if ((n - 2) in memo.keys()):
            nm2 = memo[n - 2]
        else:
            nm2 = fibonacciMemo(n - 2, memo)
            memo[n - 2] = nm2
        return nm1 + nm2
```

```
def fibonacci(n):
    if (n <= 1):
        return n
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)

>>> fibonacci(40)
102334155                    #Vreme: 41.27s

>>> fibonacciMemo(40)
102334155                    #Vreme: 9.4*10⁻⁵s

>>> fibonacciMemo(200)
280571172992510140037611932413038677189525

# Kod korišćenjem memoizacije je ograničen
# dubinom rekurzije, ali za uspešno
# određivanje 3450-og broja mu je potrebno
# 0.00545s (721 cifra)
```

Lambda izrazi



- Bazira se na konceptima pozajmljenim iz matematike
- Kreiranjem lambda izraza, kreira se expression, koji dalje može da se koristi nad raznim parametrima
- Najlakši način da se u Python programskom jeziku kreiraju funkcije koje su čiste je korišćenjem ključne reči lambda

```
>>> kvadrat = lambda x: x ** 2
>>> kvadrat(10)
100
```

Lambda izrazi - primeri

```
def kvadrat(x):
    return x ** 2

kvadratL = lambda x: x ** 2

>>> (lambda x: x ** 2)(10)
100
>>> kvadrat(10)
100
>>> kvadratL(10)
100
```

```
def dodaj(x, y):
    return x + y

dodajL = lambda x, y: x + y

>>> (lambda x, y: x + y)(10, 20)
30
>>> dodaj(10, 20)
30
>>> dodajL(10, 20)
30
```

Lambda izrazi - primeri

- Može da se sastoji od samo jednog izraza, iako je moguće da on bude u više linija

```
>>> (lambda x:
        (x if x % 2 == 0 else "Neparan broj"))(2)
2
```

- Moguće je koristiti i uslove.
  - x na početku predstavlja povratnu vrednost ukoliko je uslov (if) ispunjen
  - Podatak koji se nalazi u else grani se vraća ukoliko uslov nije ispunjen
  - Moguće je vratiti podatke različitih tipova, kao u prethodnom primeru

Lambda izrazi - primeri

- Takođe, moguće je koristiti i logičke operatore `and`, `or` i `not`

```
>>> (lambda x:
        x % 2 == 0 and "Paran" or "Neparan")(8)
'Paran'
```

- Ovaj izraz se bazira na malom triku. Ukoliko je uslov `x % 2 == 0` ispunjen, prelazi se na proveru da li je i sledeći uslov (nakon `and`) takođe `True`. Ukoliko jeste, nebitna je vrednost nakon `or` pa se zato vraća rezultat koji ispunjava uslov (`x % 2 == 0 and "Paran"`), što je `"Paran"`. Ukoliko nije ispunjen, onda sve zavisi od nastavka izraza (nakon `or`) pa se zato taj „uslov“ i vraća, što je u ovom slučaju `"Neparan"`
- Takođe, uslov ne mora da bude logički, kao u prethodnom primeru, već može da bude i `x % 2`. U tom slučaju, ukoliko je vrednost koju izraz vraća `0`, onda se tretira kao `False`, u ostalim slučajevima je `True`. Takođe, `False` vrednost imaju i prazan string i prazne kolekcije (tuple, list, dictionary), vrednost `None`, kao i sam literal `False`, dok sve ostale vrednosti se posmatraju kao `True`

Lazy evaluation (lenjo izračunavanje)

- Strategija kojom se omogućava da se objekti ne kreiraju (uglavnom kada se radi o kolekcijama), sve do momenta dok njihova vrednost nije potrebna
- Na ovaj način se štedi memorija, ukoliko se pravilno upotrebljava i vreme izvršenja programa se smanjuje
- Python podržava lenjo izračunavanje
  - Transformacija kolekcija u iteratore se vrši funkcijom iter(...)
- Negativne strane:
  - Pristupanje kolekciji više puta, pronalazi vrednost svaki put. U tom slučaju bolje je prevesti kolekciju u memorijsku
  - Pristupanje n-tom elementu kolekcije se vrši pozivanjem next(...) metode, sve dok se ne preuzme potrebna vrednost. Ovaj pristup nije previše efikasan

Lazy evaluation - primeri

```
def metoda():
    for i in range(1, 100):
        if (i % 2 == 0):
            yield i
```

```
>>> for x in metoda():
...     print(x , end = " ")
```

2 4 6 8 10 12 14 16 18 ... 98

```
# end = " " u print metodi znači
# da će se na kraju štampe, umesto
# novog reda naći string koji
# smo prosledili kroz end
```

- Ključna reč `yield` se koristi da bi se iz metode vratio specijalni tip generator, a ne kolekcija
- To omogućava da se svaki element pribavlja u onom trenutku kada bude bio potreban

```
>>> generator = metoda()
>>> print(next(generator), end=" ")
>>> print(next(generator), end=" ")
>>> print(next(generator), end=" ")
```

2 4 6

```
# Metoda next() nam vraća sledeći element
```

Curry, Uncurry, Compose

- "Currying" - tehnika koja je dobila naziv po imenu matematičara Haskela Karija, je transformisanje funkcije koja ima više argumenata u više poziva funkcija sa po jednim argumentom

```
def dodaj(a, b, c):
    return a + b + c

curry = lambda a: lambda b: lambda c: dodaj(a, b, c)

>>> curry(1)(2)(3)
6
```

- Korisno zbog pojednostavljivanja funkcija, kao i kombinacije sa drugim funkcijama

Curry, Uncurry, Compose

- Drugi način:


```
def g(x):
    def h(y):
        def i(z):
            f(x, y, z)
        return i
    return h
```

- Inverzna transformacija se naziva Uncurry

```
uncurry = lambda x, y, z: curry(x)(y)(z)

>>> uncurry(1, 2, 3)
6
```

Curry, Uncurry, Compose

- *args i **kwargs
- Ukoliko je broj argumenata koji prosleđujemo funkciji nepoznat, moguće je koristiti *args i **kwargs (naziv može da se razlikuje, *a i **k je takođe validno)
- *args nam omogućava da prikupimo listu argumenata kao listu objekata
- **kwargs se razlikuje zato što omogućava prikupljanje argumenata kao rečnika (key, value par)

Curry, Uncurry, Compose

```
def funkcija(*a, **k):
    print("args: ", a)
    print("kwargs: ", k)


>>> funkcija('Jedan', 'Dva', 'Tri', first="Prvi", second="Drugi")
args: ('Jedan', 'Dva', 'Tri')
kwargs: {'first': 'Prvi', 'second': 'Drugi'}
```

- Ovaj način čitanja argumenata može da nam posluži u automatskom prevođenju funkcije, bez obzira na njen broj parametara

Curry, Uncurry, Compose

```
def curry(func, *args, **kwargs):
    if (len(args) + len(kwargs)) >= func.__code__.co_argcount:
        return func(*args, **kwargs)
    return (lambda *x, **y: curry(func, *(args + x), **dict(kwargs, **y)))

@curry                          # Dekorator koji "pakuje" funkciju u poziv curry funkcije
def add(x, y, z):
    return x + y + z

curryVerzija = curry(add)

>>> print(add(10)(20)(30))
>>> print(curryVerzija(10)(20)(30))
60
60
```

Pojašnjenje - curry

- Parametri
  - func – funkcija koja se prosleđuje (preko atributa ili direktno)
  - *args – lista prostih argumenata (u ovom primeru samo će oni i biti potrebni)
  - **kwargs – lista dictionary argumenata (ukoliko ima argumenata koji imaju default vrednost)
    - Primer: def metoda(argument = "Default vrednost"): ...
- Prvi uslov
  - Proverava se da li je broj argumenata (sabiramo obične i default atribute) >= od broja argumenata koje funkcija prihvata. Veće je tu zbog slučaja da smo prosledili više argumenata od potrebnog broja. Ukoliko ih ima više, doći će do greške, zato što funkcija add vraća int, a on ne može da se zove kao funkcija. Da ne bi došlo do greške, moguće je obuhvatiti slučaj > posebnim uslovom, koji će da vrati grešku (objašnjenje da ne sme da bude više parametara)
  - Ukoliko je uslov ispunjen, imamo dovoljno argumenata, pa možemo da pozovemo originalnu funkciju (func). To i radimo u return func(*args, **kwargs)

Pojašnjenje - curry

- Ostalo
  - Ukoliko uslov nije ispunjen, dolazimo do poslednje linije u kodu
  - lambda *x, **y je funkcija koja preuzima listu argumenata, kao i curry funkcija (neograničen broj)
    - To znači da može da bude i više od jednog, nismo ograničeni na jedan
  - Pozivamo rekurzivno funkciju sa novim argumentima, na listu dodajemo x, a na dictionary dodajemo y.
    - Nakon toga ponovo pozivamo, sa novim argumentima i proveravamo kao i prvi put
- Šta ako prosledimo add(10), bez dodatnih argumenata. Dobićemo kao rezultat lambda izraz koji je kreiran u poslednjoj liniji. Znači funkciju, koja sada od nas očekuje jedan argument manje.
  - add(10)(20) nam takođe vraća funkciju (lambda), kojoj nedostaje još samo jedan argument. Dosadašnja lista (10, 20) se pamti, a čeka se sledeći, treći parametar

Pojašnjenje – curry – primeri

```
>>> add(10)
<function __main__.curry.<locals>.<lambda>(*x, **y)>
# Povratni tip nam je lambda izraz, koji sada očekuje jedan parametar manje
>>> add(10, 20)(30)
60
# Takođe validan izraz, koji vraća isti rezultat. Prvi poziv ima 2 argumenta
# dok drugi ima još jedan koji nedostaje
>>> add(10, 20)
<function __main__.curry.<locals>.<lambda>(*x, **y)>
# Takođe vraća funkciju kojoj sada nedostaje jedan argument
>>> add(10)(20)(30)(40)
TypeError: 'int' object is not callable
# int, koji je dobijen posle prva 3 argumenta, ne može da se poziva kao f-ja
```

Curry, Uncurry, Compose

- Curry funkcija prihvata funkciju koju treba transformisati, kao i argumente, u obliku liste ili rečnika
- Ukoliko broj argumenata odgovara broju argumenata prosleđenom od strane korisnika, onda se funkcija izvršava i šalje rezultat
- Ukoliko ne, jedan od argumenata se dodaje korišćenjem lambda izraza (nove funkcije), i funkcija curry se rekurzivno poziva za funkciju sa starim parametrima, uz dodatak novih, kreiranih putem lambda izraza

Compose

```
add = lambda x: x + 10
subtract = lambda x: x - 2
multiply = lambda x: x * 10
divide = lambda x: x // 2

def compose(*funcs):
    head, *tail = funcs
    if (len(tail) >= 2):
        return lambda x: compose(*tail)(head(x))
    else:
        [first] = tail
        return lambda x: first(head(x))

>>> compose(add, multiply, add, subtract, divide)(10)
104
combination = lambda x: divide(subtract(add(multiply(add(x)))))
>>> combination(10)            # Kombinacija korišćenjem lambda izraza
104
```

Curry, Uncurry, Compose

- Kompozicija funkcija može da bude korisna u velikom broju slučajeva, naročito kod ulančavanja funkcija
  - Transformacija jedinica, vremena, temperature, valuta, su samo neki primeri koji mogu da se pojednostave na ovaj način

```
head, *tail = funcs
# head i *tail omogućavaju da se funcs lista podeli na prvi element (head) i
# na ostatak liste, koji se smešta u tail
*tail
# Starred expression (*) se koristi u bilo kom kontekstu (kao i u primeru
# iznad) za otpakivanje liste
[first] = tail
# Samo jedan element liste može da se otpakuje u promenjivu korišćenjem
# uglastih zagrada (sintakse slične nizu)
```