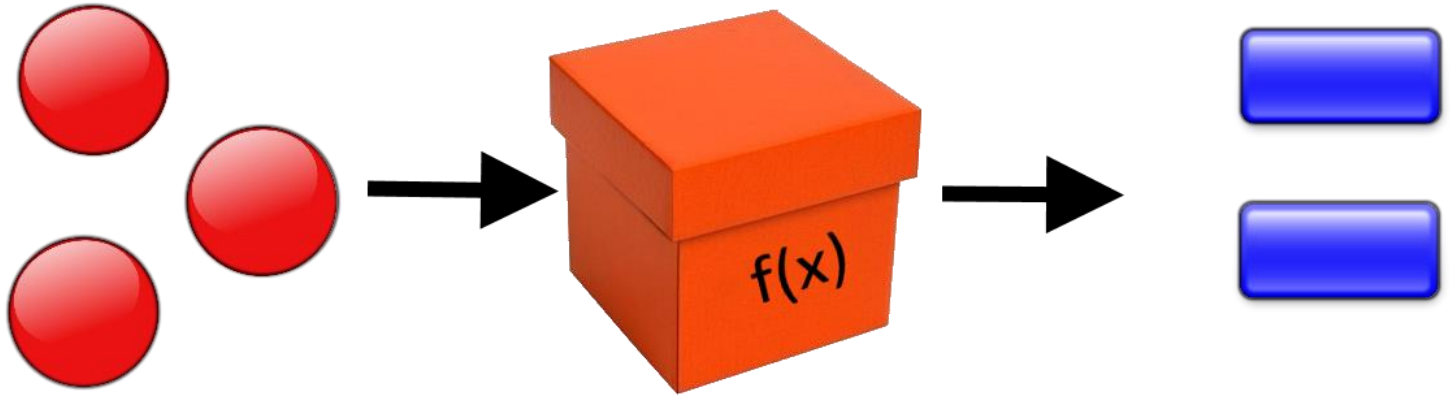
Katedra za računarstvo

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 - rekurzija

- 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

Repna rekurzija – tail recursion

```
# Verzija sa rekurzijom glave (head recursion)
def factorial(n):
  if n == 0:
    return 1
  else:
    return factorial(n - 1) * n

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

Funkcije - rekurzija

- 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

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'
>>> stampaj("Poruka")
Poruka
```

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

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

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 parametra

Monad

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

```
def kombinacijaFunkcija(drugaFja, prvaFja, broj: int) -> (int, str):
    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

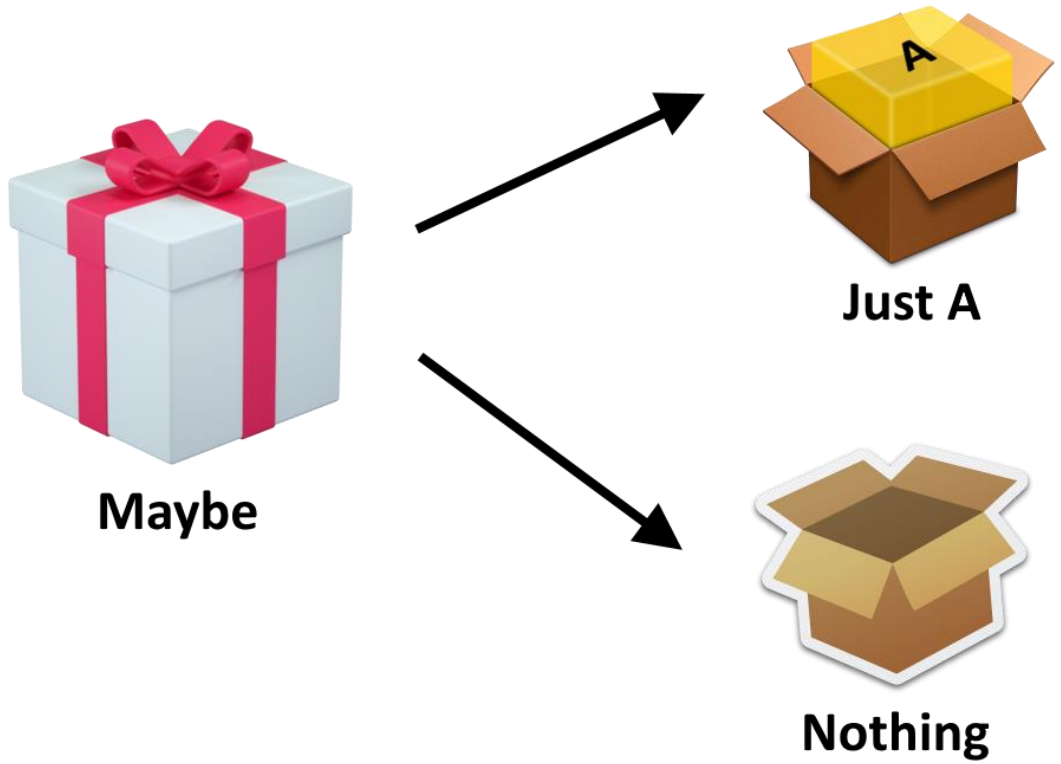
- Rešenje su 2 funkcije:
  - Funkcija unit, koja broj koji se prosleđuje na početku transformiše u tuple podatak i dodaje mu string koji će da pamti brojeve
  - Funkcija bind koja će da prihvati funkciju koju treba izvršiti i podatak u odgovarajućem obliku (bez obzira da li ga je kreirao unit ili neki 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 će se nadovezivati novi brojevi
- 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)) -> (int, str):
   rez = fja(prethodni[0])
   return (rez[0], prethodni[1] +
            (", " if prethodni[1] else "") +
              rez[1])

def unit(broj: int) -> (int, str):
   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 greške
+ 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 pratiti u funkcionalnom programiranju. Ono se prenosi sa izlaza jedne na ulaz druge funkcije

Druge optimizacije

- Memoizacija (memoization)
  - 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 (lazy evaluation)
  - 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)
```

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 kao i vreme izvršenja programa koje 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

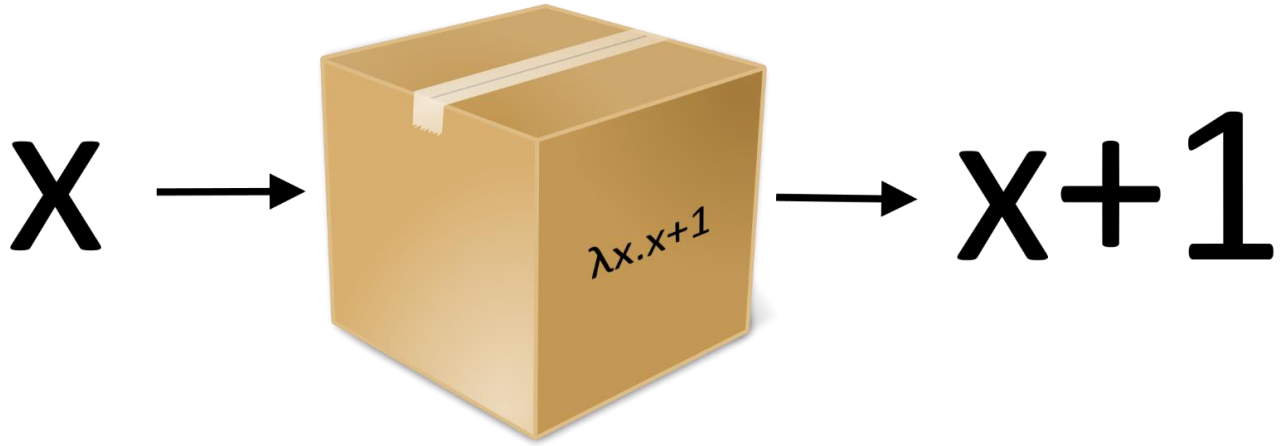
```
def metoda():
    for i in range(1, 100):
        if (i % 2 == 0):
            yield i
```

```
>>> 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
# Funkcija next() nam vraća sledeći element
```

Lambda izrazi



- Bazira se na konceptima pozajmljenim iz matematike
- Kreiranjem lambda izraza, kreira se expression, koji dalje može da se koristi u kombinaciji sa raznim argumentima
- 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 se sve ostale vrednosti posmatraju kao `True`

min, max

- **min**(iterable *[, default=obj, key=func]*) -> *value*
- **min**(a, b, *c *[, key=func]*) -> *value*
- **max**(iterable *[, default=obj, key=func]*) -> *value*
- **max**(a, b, *c *[, key=func]*) -> *value*

Parametar	Opis
iterable *(obavezan)*	Kolekcija podataka (string, lista, tuple,...)
default *(opcioni)*	Vrednost koju funkcija vraća ukoliko je kolekcija prazna
key *(opcioni)*	Funkcija sa jednim parametrom

Parametar	Opis
a, b, *c *(obavezan)*	Više parametara (najmanje 2) za koje se traži minimum ili maksimum
key *(opcioni)*	Funkcija sa jednim parametrom

min, max

- min i max funkcije prihvataju kolekciju podataka ili više parametara koji se tretiraju kao pojedinačni elementi i vraćaju najmanji/najveći
- Ukoliko se radi o listi elemenata, zato što ona može da bude prazna, postoji *default* parametar kojim se funkciji prosleđuje vrednost koju treba da vrati ukoliko nema elemenata za poređenje. Ukoliko ne koristimo parametar default, prazna kolekcija će vratiti **ValueError**.
- Parametar *key* prihvata funkciju kojoj se svaki element za poređenje prosleđuje, a upoređivanje se vrši na osnovu vrednosti koju je ona vratila

```
>>> min([(6, 1), (3, 3), (7, 2), (2, 5)], key=lambda x: x[1])
(6, 1)
```

min, max

- Drugi oblik funkcija min i max prihvata listu argumenata koji se porede direktno
- Ovi argumenti takođe mogu da budu kolekcije

```
>>> max((6, 1), (3, 3), (7, 2), (2, 5), key=lambda x: x[1])
(2, 5)
>>> min([], default=0)
0
>>> max(["1234", "12", "123"], key=len)
'1234'
```

sorted

- **sorted**(iterable, *[key=None, reverse=False]*) -> *iterable*

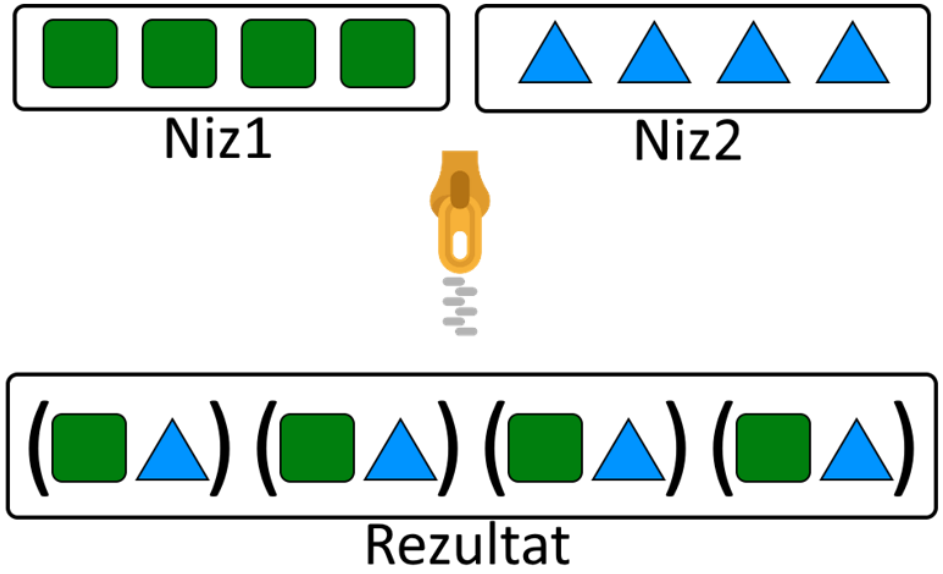
- sorted funkcija prihvata kolekciju podataka, sortira je i vraća novu listu sortiranu u rastući redosled
- Kao i kod min i max funkcija, moguće je koristiti opcioni *key* parametar čija vrednost treba da bude funkcija sa jednim parametrom
- Osim key parametra, moguće je koristiti i opcioni *reverse* parametar koji kolekciju vraća u opadajućem redosledu
- *sorted* ne menja prosleđenu kolekciju (funkcionalna je). Postoji i *sort* verzija koja nije funkcionalna i zove se nad objektom

sorted

```
>>> sorted([(6, 1), (3, 3), (7, 2), (2, 5)], key=lambda x: x[1])
[(6, 1), (7, 2), (3, 3), (2, 5)]
>>> sorted([1, 2, 3, 4, 5], reverse=True)
[5, 4, 3, 2, 1]
>>> sorted(("Jedan", "Dva", "Tri", "Četiri"), key=len)
['Dva', 'Tri', 'Jedan', 'Četiri']
>>> a = [(1, 4), (4, 2, 6), (3, 5)]
>>> a.sort(key=lambda x: x[1], reverse=True)
>>> a # vrednost a je izmenjena
[(3, 5), (1, 4), (4, 2, 6)]
```

zip

- **zip**(iter1 *[, iter2 [...]]*) -> *zip object*



- Koristi se za spajanje 1 ili više kolekcija
- zip funkcija ima specijalnu povratnu vrednost (zip object) koja predstavlja iterator tuple tipova
- Nije potrebno proslediti funkciju, već se spajanje vrši automatski
  - Prvi element iz prve kolekcije se spaja sa prvim elementom druge kolekcije...

```
>>> list(zip([1, 2, 3], ["1", "2", "3"]))
[(1, '1'), (2, '2'), (3, '3')]
```

Curry, Uncurry, Compose

- "**Currying**" - tehnika koja je dobila naziv po imenu matematičara Haskela Karija (eng. Haskell Curry), je transformisanje funkcije koja ima više argumenata u više poziva funkcija sa po jednim parametrom (ili više parametara)

```
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):
            return f(x, y, z) # f je u našem primeru add
        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 su takođe validni)
- *args nam omogućava da prikupimo listu argumenata kao listu objekata
- **kwargs se razlikuje zato što omogućava prikupljanje default 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 None
    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 dekoratora 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(parametar = "Default vrednost"):...`
- Prvi uslov
  - Proverava se da li je broj argumenata (sabiramo obične i default argumente) >= broju parametara 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 int ne može da se poziva 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, rekurzivno pozivamo, sa novim argumentima curry funkciju 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
```

```
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
  - 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
# Elementi liste mogu da se otpakuju u promenjive korišćenjem
# uglastih zagrada (sintakse slične nizu)
```

Uključivanje biblioteka i modula

- Python je programski jezik poznat po velikom broju biblioteka
- Biblioteke / module je moguće uključiti na više načina, u zavisnosti od potrebnih funkcionalnosti

```
# Uključivanje celokupnog modula
import math
# Uključivanje samo jedne funkcije
from math import pow
# Uključivanje svih funkcija i konstanti
from math import *
# Uključivanje funkcije sa promenjenim nazivom
from math import pow as stepen
# Postoje i relativne import naredbe (from . import funkcija), ali nećemo ih koristiti
```

Instalacija biblioteka

- Biblioteke mogu da se instaliraju putem "pip" komande (Package Installer for Python). Paketi se instaliraju sa Python Package Index-a. Komanda se koristi iz terminala po izboru (npr. CMD-a)
- pip install tail-recursive
  - Biblioteka koja omogućava reprnu rekurziju, uz minimalne modifikacije koda

```
from tail_recursive import tail_recursive

@tail_recursive
def factorial(n):                          # Verzija funkcije sa rekurzijom glave
    if n == 0:
        return 1
    return n * factorial.tail_call(n - 1)  # tail_call nam omogućava optimizaciju

>>> print(factorial(1000))                 # Broj rekurzivnih poziva više nije ograničen
402387260077093773543702433923003985719374864...
```

Instalacija biblioteka

```
from tail_recursive import tail_recursive

@tail_recursive
def factorialRR(n, acc = 1):                    # Verzija funkcije sa rekurzijom repa
    if n == 0:
        return acc
    return factorialRR.tail_call(n - 1, acc * n) # tail_call nam omogućava optimizaciju

>>> print(factorialRR(1000))                    # Broj rekurzivnih poziva više nije ograničen
402387260077093773543702433923003985719374864...
```

- Razlika između dve prethodne funkcije više nije dubina do koje je moguće vršiti rekurziju, zato što u oba slučaja broj rekurzivnih poziva nije ograničen, već brzina kojom će se kod izvršiti, koji je nešto brži u slučaju kada funkciju implementiramo kroz repnu rekurziju

map, filter

- Funkcionalno programiranje je najjednostavniji način za obradu kolekcija
- Koristi se set predefinisanih funkcija, koje prihvataju korisničku funkciju koja se koristi za obradu svakog od elemenata kolekcije, koja se prosleđuje kao parametar
- map se koristi da se korisnička funkcija izvrši nad svakim elementom i vrati transformisani element u rezultujuću kolekciju
- filter izvršava funkciju nad svakim elementom, ali u rezultujuću kolekciju smešta samo one elemente koji prosleđeni uslov zadovoljavaju
- `map(function, sequence[, sequence, ...]) -> map object`
- `filter(function or None, iterable) -> filter object`

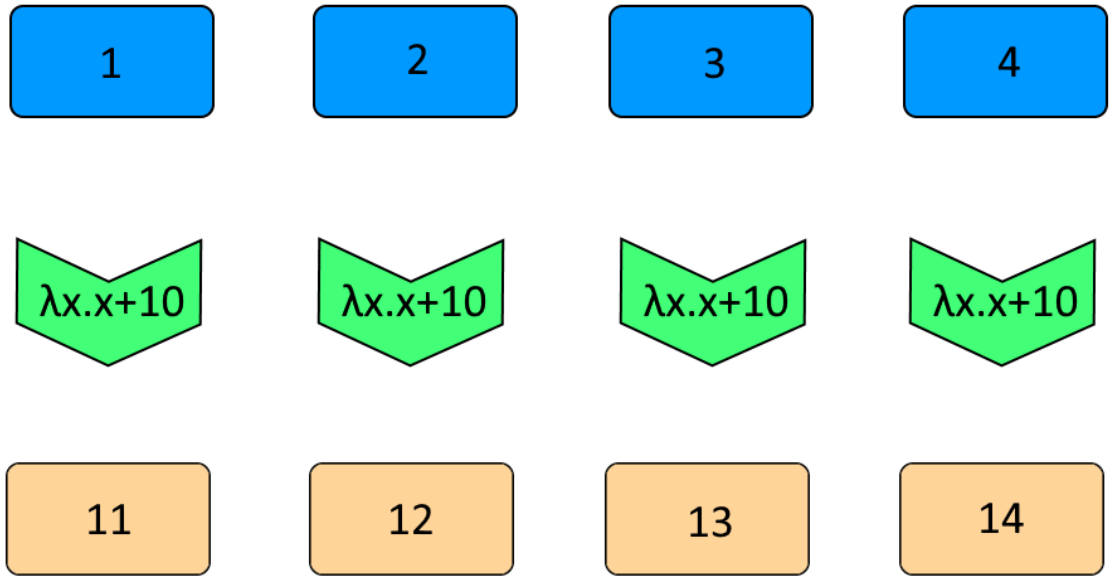
map

```
>>> list(map(
        lambda x: x + 10,
        [1, 2, 3, 4]))
[11, 12, 13, 14]
```

- Svaki element kolekcije koji se prosleđuje funkciji kao drugi parametar se koristi za kreiranje rezultujuće kolekcije primenom funkcije koja se prosleđuje kao prvi parametar.
- Može da ima više od jedne ulazne kolekcije.

```
>>> list(map(
        lambda x, y: x + y,
        [1, 2, 3, 4], [1, 2, 3, 4]))
[2, 4, 6, 8]
```

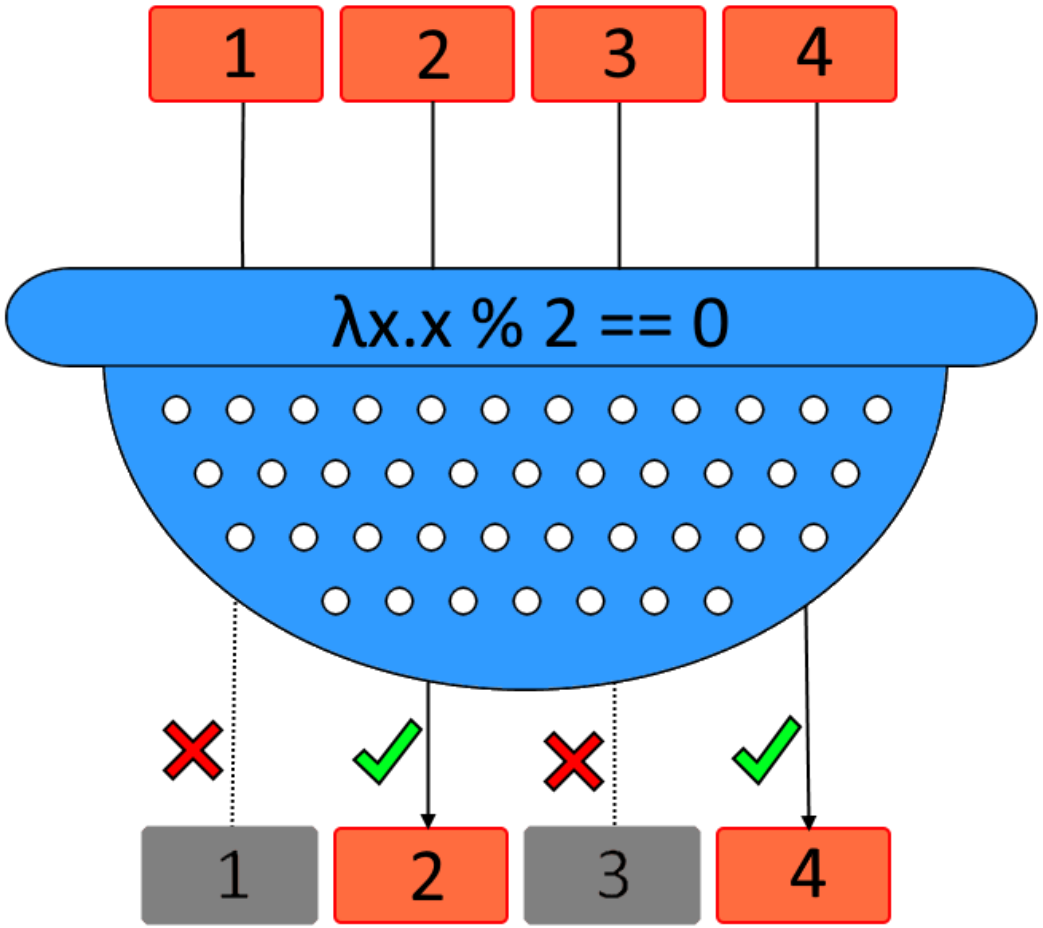
- map vraća specijalni objekat tipa map, koji je neophodno prevesti u listu



filter

```
>>> list(filter(
        lambda x: x % 2 == 0,
        range(1, 21)))
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

- I kod funkcije filter, niz se prosleđuje na isti način, ali se funkcija razlikuje. Funkcija koju filter koristi mora da vraća logičku vrednost (sve što vrati će se tretirati kao logička vrednost)
- Takođe je neophodno izvršiti prevođenje u listu



Modul **functools** - funkcija **reduce**

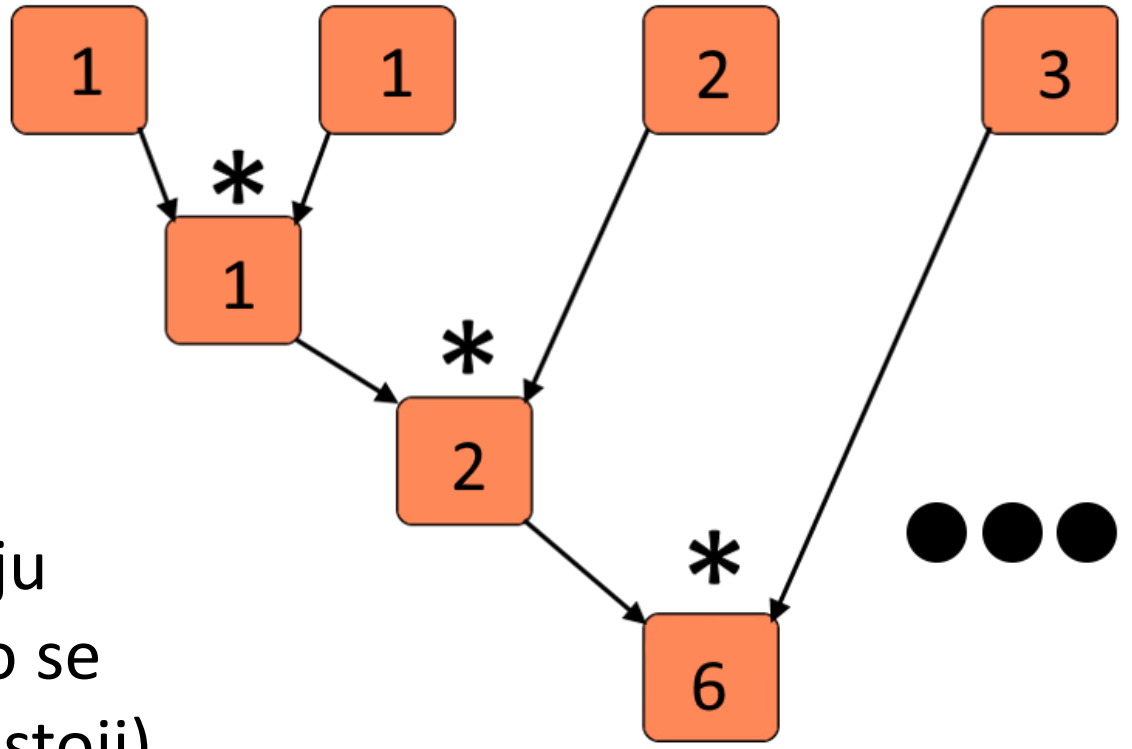
- `functools.reduce(function, sequence[, initial]) -> value`
- Jedan od modula, koji sadrži funkcije za rad sa kolekcijama je functools
  - Kao što sam naziv govori ovo je modul funkcionalnih operatora

```
>>> from functools import reduce
>>> reduce(lambda x, y: x + y, range(1, 101))
5050
```

- Funkcija reduce se koristi da kolekciju elemenata transformiše u rezultat koji se sastoji od jedne vrednosti
- Prvi parametar je funkcija koja prihvata dve vrednosti, a ima povratnu vrednost koja je jedna vrednost
- Drugi parametar je kolekcija nad kojom treba primeniti reduce funkciju

Modul **functools** - funkcija **reduce**

```
>>> reduce(lambda x, y: x * y, range(1, 11), 1)
3628800
```



- Ukoliko je potrebno, moguće je koristiti i treći parametar, početnu vrednost akumulatora (initial) u koju se smešta konačan rezultat (ukoliko se ne koristi, inicijalna vrednost ne postoji)

Modul **functools** - funkcija **cache**

- `functools.cache(user_function) -> lru cache wrapper`

- Još jedna funkcija u modulu functools može da olakša **memoizaciju**. To je funkcija cache.
- Može da se koristi na 2 načina:
  - functools.cache(funkcija) ili
  - Korišćenjem dekoratora @cache

```
# Ukoliko nema dekoratora
fibonacci = cache(fibonacci)
>>> print(fibonacci(1300))
21599680283161715807847052...
```

```
from functools import cache
@cache
def fibonacci(n):
    if (n <= 1):
        return n
    else:
        return (fibonacci(n - 1) +
                fibonacci(n - 2))
>>> print(fibonacci(1300))
21599680283161715807847052...
```

Modul **functools** - funkcija **lru_cache**

- `functools.lru_cache(user_function) -> *lru cache wrapper*`
- `functools.lru_cache(maxsize=128, typed=False) -> *lru cache wrapper*`

- Pored cache funkcije, postoji i lru_cache.
- Razlika između ovih funkcija je u tome što je kod funkcije lru_cache moguće proslediti i argumente
  - maxsize, koji onemogućava pamćenje više vrednosti od vrednosti ovog argumenta i
  - typed, koji, ukoliko je True, identične vrednosti različitih tipova podataka (npr. float i int), pamti kao različite unose u rečniku
- Korišćenje funkcije cache proizvodi minimalno brži kod, ali funkcija lru_cache može da bude jako korisna u slučaju velikih rečnika

```
from functools import lru_cache

fibonacci = lru_cache(32, True)(fibonacci)

@lru_cache(maxsize=32, typed=True)
def fibLRUD(n):
    if (n <= 1):
        return n
    else:
        return (fibLRUD(n - 1) + fibLRUD(n - 2))

>>> print(fibonacci(1300))
21599680283161715807847052...
>>> print(fibLRUD(1300))
21599680283161715807847052...
```

Modul **functools** - funkcija **partial**

- `functools.partial(func, *args, **keywords) -> partial`

- Funkcija partial, koja se nalazi u functools modulu, može da kreira novu funkciju, od prosleđene funkcije i liste argumenata.
- Funkcija se uvek poziva sa prosleđenim argumentima, a ostali (izostavljeni) argumenti se mogu naknadno proslediti
- Slično "Curry"

```
from functools import partial
def metoda(p, q):
    print(f"{p}, {q}")
>>> proba = partial(metoda, 10)
>>> proba(50)
10, 50
>>> proba = partial(metoda, 2, 6)
>>> proba()
2, 6
```

Modul **operator**

- Modul operator sadrži sve osnovne **matematičke i logičke operatore**, kao i operatore **setitem**, **delitem** i **getitem**, koji manipulišu slice objektima (kreiranim putem slice funkcije)
- Svi ovi operatori su implementirani kao funkcije, pa se mogu prosleđivati drugim funkcijama

```
import operator

tail = slice(1, None)      # Indeks elemenata od 2. pa do poslednjeg
>>> print(operator.getitem([1, 2, 3], tail))
[2, 3]
>>> print(reduce(operator.add, [1, 2, 3]))
6
```

Modul **itertools**

- Modul koji sadrži funkcije za kreiranje iteratorskih podataka
- Dele se na:
  1. Beskonačne iteratore
  2. Iteratore koji se prekidaju kada se prva kolekcija završi
  3. Kombinatoričke iteratore

1. U ovu kategoriju spadaju: count, cycle i repeat

```
from itertools import *    # Pre korišćenja f-ja, neophodno uključiti modul
```

- `itertools.count(start=0, step=1) -> count iterator`

```
>>> list(count(10, 3))
10, 13, 16, 19 .... ∞
```

Modul **itertools**

- `itertools.`**`cycle`**`(iterable) -> `*`cycle iterator`*

```
>>> list(cycle([1, 2, 3]))
[1, 2, 3], [1, 2, 3] .... [1, 2, 3]...
```

- `itertools.`**`repeat`**`(object`*`[, times]`*`) -> `*`repeat iterator`*

```
>>> list(repeat([1, 2], 10))
[1, 2], [1, 2] ... [1, 2] (10 puta)
```

  - Ukoliko se drugi parametar izostavi, onda ∞ puta

Modul **itertools**

2. Iteratori koji se prekidaju kada se prva kolekcija završi

- itertools.**accumulate**(iterable*[, func, initial=None]*) -> *accumulate*

```
>>> print(list(accumulate(range(1, 11), lambda x, y: x * y, initial=1)))
[1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

- itertools.**chain**(*iterables) -> *chain*

```
>>> print(list(chain("ABC", "DEF")))
['A', 'B', 'C', 'D', 'E', 'F']
```

- itertools.**compress**(data, selectors) -> *compress*

```
>>> print(list(compress("ABCDEF", [1, 1, 0, 1, 0, 1])))
['A', 'B', 'D', 'F']
```

- itertools.**dropwhile**(predicate, iterable) -> *dropwhile*

```
>>> print(list(dropwhile(lambda x: x < 5, [1, 3, 5, 1, 12, 4, 15])))
[5, 1, 12, 4, 15]
```

Modul **itertools**

- itertools.**filterfalse**(predicate, iterable) -> *filterfalse*

```
>>> print(list(filterfalse(lambda x: x < 5, [1, 3, 5, 1, 12, 4, 15])))
[5, 12, 15]
```

- itertools.**groupby**(iterable, key=None) -> *groupby*

```
>>> for k, g in groupby('AAAABBBCCDAABBB'):
...     print((k, len(list(g))), end=" ")
('A', 4) ('B', 3) ('C', 2) ('D', 1) ('A', 2) ('B', 3)
```

- itertools.**islice**(iterable, stop) -> *islice*
- itertools.**islice**(iterable, start, stop[, step]) -> *islice*

```
>>> print(list(islice("ABCDEF", 1, 4, 2)))       # start, stop, step
['B', 'D']
```

- itertools.**pairwise**(iterable) -> *pairwise*

```
>>> print(list(pairwise("ABCD")))
[('A', 'B'), ('B', 'C'), ('C', 'D')]
```

Modul **itertools**

- itertools.**starmap**(function, iterable) -> *starmap*

```
>>> print(list(starmap(lambda x, y: x ** y, [(2, 5), (2, 3)])))
[32, 8]          # Primenjuje lambda izraz nad tuple tipovima u kolekciji
```

- itertools.**takewhile**(predicate, iterable) -> *takewhile*

```
>>> print(list(takewhile(lambda x: x < 20, count(10, 3))))
[10, 13, 16, 19]
```

- itertools.**tee**(iterable, n=2) -> *tee*

```
>>> for x in list(tee(range(1, 10), 2)):  # Vraća n ponavljanja prosleđene kolekcije
...     print(list(x))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- itertools.**zip_longest**(*iterables, fillvalue=None) -> *zip_longest*

```
>>> print(list(zip_longest('ABCD', 'xy', fillvalue='-'))) # Slično zip, cela kolekcija
[('A', 'x'), ('B', 'y'), ('C', '-'), ('D', '-')]
```

Modul **itertools**

3. Kombinatorički iteratori

- itertools.**product**(*iterables, repeat=1) -> *product*

```
>>> print(list(product("AB", "AB")))
[('A', 'A'), ('A', 'B'), ('B', 'A'), ('B', 'B')]
```

- itertools.**permutations**(iterable, r=None) -> *permutations*

```
>>> print(list(permutations("ABC", 2)))                # r=2, permutacije dužine 2
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
```

- itertools.**combinations**(iterable, r) -> *combinations*

```
>>> print(list(combinations("ABCD", 2)))               # kombinacije dužine 2
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]
```

- itertools.**combinations_with_replacement**(iterable, r) -> *combinations with replacement*

```
>>> print(list(combinations_with_replacement('ABC', 2))) # kombinacije sa ponavljanjem dužine 2
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('B', 'B'), ('B', 'C'), ('C', 'C')]
```

List comprehensions

- Comprehension sintaksa u Python-u omogućava jednostavno kreiranje kolekcija (lista, tuple, dictionary), na osnovu postojeće kolekcije

```
newlist = []

for x in range(1, 10):
    newlist.append(x)

>>> print(newlist)
[1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> print([x for x in range(1, 10)])
[1, 2, 3, 4, 5, 6, 7, 8, 9]
# Sintaksa podseća na petlju. Jedina razlika je što se promenjiva upisuje
# direktno u niz
>>> print(list(((x, y) for x in range(1, 3) for y in range(1, 3))))
[(1, 1), (1, 2), (2, 1), (2, 2)]                    # Dvostruka petlja
```

Set comprehensions, Dictionary comprehensions

- Ista sintaksa može da se koristi i za ostale tipove kolekcija
- Set

```
>>> { 10 for x in range(1, 11) }
{10}    # Iako smo upisali 10 elemenata, svaki ima istu vrednost
```

- Dictionary

```
>>> { x: chr(x) for x in range(97, 97 + 26) }
{97: 'a', 98: 'b', 99: 'c', 100: 'd', 101: 'e', ..., 121: 'y', 122: 'z'}
```

- Uslovi

```
list((x if x % 2 == 0 else 100 for x in range(1, 11)))
[100, 2, 100, 4, 100, 6, 100, 8, 100, 10]
# Sintaksa je nešto drugačija, koriste se obične zagrade. Uslovi su na
# početku, x će biti upisano ukoliko je uslov ispunjen, ukoliko ne, prelazi se na
# else, pa će biti upisana vrednost 100
```

Comprehensions kao "zamena" za map, filter

- Sličnosti sa map i filter

```
>>> print(list(map(lambda x: x + 10, range(1, 10))))
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> print(list([x + 10 for x in range(1, 10)]))
[11, 12, 13, 14, 15, 16, 17, 18, 19]
# Jednostavan list comprehension može da zameni map funkciju iznad

>>> print(list(filter(lambda x: x < 5, range(1, 10))))
[1, 2, 3, 4]
>>> print(list([x for x in range(1, 10) if x < 5]))
[1, 2, 3, 4]
# Uslov je ovde pomeren na kraj izraza, nakon for petlje. Uslov se nalazi
# ispred, kada obavezno koristimo i else, pa želimo da odlučimo između dve
# vrednosti. Kada se nađe iza, ukoliko uslov nije ispunjen, element niza se
# ignoriše
```

Comprehensions kao "zamena" za reduce

- Nije uvek moguće zameniti poziv reduce funkcije

```
listT = [[1, 2, 3], [2, 3, 4], [3, 4, 5]]

>>> print(reduce(lambda x, y: x + y, listT))
[1, 2, 3, 2, 3, 4, 3, 4, 5]

>>> print([item for sublist in listT for item in sublist])
[1, 2, 3, 2, 3, 4, 3, 4, 5]
# Pristup je potpuno drugačiji. reduce funkcija poziva + operator nad 2 liste
# [1, 2] + [3, 4] = [1, 2, 3, 4].
# Sa druge strane, comprehension prolazi kroz listu lista, a zatim u drugom
# prolazu i kroz podlistu, pa zato se u item elementu nalazi svaki pojedinačni
# element niza
```

Lazy evaluation – Fibonacci (pravi funkcionalni način)

```
from itertools import islice
def iterate(func, param):                    # Kreira generator. Vraća vrednost samo kada je potrebna
    while True:
        yield param
        param = func(param)                  # Kreira sledeći parametar na osnovu prethodnog

def next_fibonacci(pair):                    # Funkcija koja računa sledeći broj na osnovu prethodna 2
    x, y = pair
    return (y, x + y)

def fibonacci_numbers():                     # Definiše sve Fibonačijeve brojeve do ∞ (Lazy)
    return (x for x, _ in iterate(next_fibonacci, (0, 1)))

list(islice(fibonacci_numbers(), 0, 10)) # islice preuzima samo brojeve od 0-tog indeksa do 10
>>> [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Regularni izrazi (Regular Expressions)

- Regularni izrazi se sastoje od specijalnog niza karaktera, koji omogućava pretragu stringova
- Python pruža podršku za rad sa regularnim izrazima kroz modul "re"

```
import re
```

- Za potrebe kreiranja "pattern"-a koji se koristi za pretragu, moguće je koristiti specijalan tip literala koji počinje slovom r

```
r'Regularni izraz'
```

- Takođe je moguće koristiti i regularni string
- Upotreba specijalnog literala je jednostavnija, naročito zbog mogućnosti korišćenja escape karaktera bez dupliranja

```
'\\d+'
r'\d+'
```

Regularni izrazi

- Modul re sadrži metode za rad sa regularnim izrazima. Ovde će biti prikazan rad samo nekih od ovih modula
- match
  - Funkcija match pretražuje string samo sa početka string-a. Ukoliko se karakteri na početku ne poklapaju sa šablonom, ne vraća se poklapanje. Ukoliko se početak poklapa sa šablonom, vraća se samo prvo poklapanje

```
>>> re.match(r"(\d+)", "123abcdef1234")
<re.Match object; span=(0, 3), match='123'>
```

- search
  - Za razliku od match funkcije, search pretražuje celokupan string. Kao i match, povratna vrednost je samo jedno poklapanje

```
>>> re.search(r"(\d+)", "123abcdef1234")    # .group() preuzima vrednost
<re.Match object; span=(0, 3), match='123'>
```

Regularni izrazi

- findall
  - Ukoliko postoji potreba za povratkom svih poklapanja, koristi se ova funkcija

```
>>> re.findall(r"(\d+)", "123abcdef1234")
['123', '1234']
```

  - Povratna vrednost u ovom slučaju je lista pronađenih vrednosti
- compile
  - Ukoliko se šablon često koristi, moguće ga je kompilovati, a zatim koristiti ponovo

```
>>> cre = re.compile(r"(\d+)")
>>> cre.findall("123ABCD1234")
['123', '1234']
```

- escape
  - Funkcija kojoj je moguće proslediti string, a koja kao rezultat vraća string kome su svi specijalni karakteri zamenjeni escape verzijama ('.' -> '\. ')...

Regularni izrazi

- Korisne konstante:
  - re.A ili re.ASCII - specijalne vrednosti (\w, \W, \b, \B, \d, \D, \s, \S), pretražuju samo ASCII vrednosti, ne Unicode
  - re.I ili re.IGNORECASE - pretraga se vrši bez obzira na veličinu slova
  - re.M ili re.MULTILINE - karakter '^' pretražuje početak teksta, ali i svake nove linije kao i karakter '$' (kraj linije ili teksta)
  - re.S ili re.DOTALL omogućava karakteru '.' da poklopi svaki karakter, uključujući i novu liniju. Bez ove konstante, nova linija se ne posmatra kao karakter

```
>>> re.findall("(.+)", "123ABCD1234\nNOV", re.IGNORECASE | re.DOTALL)
['123ABCD1234\nNOV']
```

Pattern matching (verzija >= 3.10.0)

- Strukturna pretraga po šablonu u Python-u se vrši korišćenjem match i case ključnih reči
- Svaki case prati šablon koji match pokušava da preklopi
- Pretraga šablona koji se poklapa sa match uslovom se vrši odozgo na dole
- Kada se prvi case poklopi, završava se match blok i prelazi se na ostatak koda
- Moguće je u šablonu postaviti više od jednog uslova

Pattern matching

```
match 'a':
    case ('A' | 'B' | 'C') as slovo:   # Više od jednog šablona, onaj koji
        print(slovo)                   # se poklapa se smešta u slovo
    case 'a' if False:                 # Pored šablona, moguće je dodati i
        print("Uslov")                 # uslov. Ukoliko nije ispunjen, case se zanemaruju
    case ('a', _):                     # Moguće je koristiti i _. Podatak
        print("Novi")                  # poklopljen sa _ se ignoriše
    case _:                            # case _ se izvršava kada nema drugog
        print("Default")               # šablona u prethodnim case koji se poklapa
```

Pattern matching

```
[prvi, drugi] = input("Unesite inicijale.")

match (prvi, drugi):
    case ('P', 'P'):  # Podatak može da bude bilo kog tipa
        print("Dobrodošli, Petar Petrović.")
    case { 'P': x, **ostali }:
        print(f"Rečnik, ključ P, vrednosti: {x}")
    case ('P', x):    # Može da bude i *x, ukoliko ima više vrednosti
        print(f'Dobrodošli, Petar {x}.')
    case _:           # Ukoliko nije pronašao ništa ranije, poziva se _
        print("Žao nam je, nema Vas na spisku.")


Dobrodošli, Petar S.  # Za ulaz PS
```

Pattern matching

- Kao što se može videti na prethodnom slajdu, osim konstanti, case može da sadrži i promenjive. Unutar case mogu da se nalaze i različiti tipovi:
  - Svi prosti tipovi
  - Tuple tip, sa neograničenim brojem vrednosti
  - Dictionary, kao i druge kolekcije

- Korišćenje pattern matching-a može da uprosti kod, kada se on sastoji od velikog broja uslova (if, elif, else grana)

- Omgućava da se komplikovani uslovi prevedu u jednostavnije (najviše u slučaju rada sa tuple, dict i drugim kolekcijama)