SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Boris Kišić

LAMBDA RAČUN S PRIMJERIMA U PROGRAMSKOM JEZIKU PYTHON

DIPLOMSKI RAD

Varaždin, 2013.

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Boris Kišić

Redoviti student

Broj indeksa: 40538/11-R

Smjer: Baze podataka i baze znanja

LAMBDA RAČUN S PRIMJERIMA U PROGRAMSKOM JEZIKU PYTHON

DIPLOMSKI RAD

Voditelj:

Doc. dr. sc. Markus Schatten

Varaždin, lipanj 2013.

Sadržaj

1. Uvod

Zapis brojeva kakvog danas koristimo u Europi je prisutan od vremena renesanse (oko 1200. godine), a jedan od prvih matematičara koji ga je koristio bio je Leonardo Fibonacci. Karakteristika našeg brojevnog sustava je mali broj znamenaka, a preuzet je od Arapa, koji su ga preuzeli od Indijaca. Nije poznato gdje je i kada taj brojevni sustav izumljen.

Notacija za izraze i jednadžbe nije postojala do 17. stoljeća, kada je Francois Viète počeo sistematično koristiti rezervirana mjesta za parametre i skraćenice za aritmetičke operatore. Do tada je bilo potrebno zapisivati sve aritmetičke operatore u obliku teksta. Prošlo je još 250 godina kada je Alonzo Church razvio notaciju za proizvoljne funkcije. Njegova notacija naziva se lambda račun (λ-račun). [7, str. 1]

Gottfried Leibniz (1646.-1716.), njemački filozof i matematičar, je imao sljedeće zamisli:

1. Stvaranje "univerzalnog jezika" pomoću kojeg će moguće biti zapisati sve moguće probleme.

2. Pronalazak metode odlučivanja koja će rješavati sve probleme zapisane pomoću univerzalnog jezika.

Ako se ograničavamo samo na matematičke probleme, prva točka Leibnizove ideje je ispunjena ako se uzme neki oblik teorije skupova formulirane u jeziku predikatne logike prvog reda. Druga točka je postala važno filozofsko pitanje, pri čemu je bilo upitno je li moguće pronaći jedinstven način za rješavanje svih problema formuliranih u univerzalnom jeziku. Problem je postao poznat pod nazivom "Entscheidungsproblem".

Nekoliko godina se činilo kako nije moguće pronaći takvu metodu, međutim problem su riješili Alonzo Church i Alan Turing, koji su to učinili na dva potpuno različita načina:

- Church je 1932. objavio formalni sustav lambda račun, te je definirao zapis izračunive funkcije pomoću tog sustava.

- Turing je 1936./1937. izumio klasu strojeva (koji se danas nazivaju Turingovim strojevima), te je definirao zapis izračunljive funkcije pomoću tih strojeva.

Također, Turing je 1936. dokazao da su oba modela jednako moćna, budući da definiraju istu klasu izračunljivih funkcija. Na konceptu Turingovog stroja temelji se Von Neumannova arhitektura računala. Konceptualno, Von Neumannova računala su zapravo Turingovi strojevi s registrima sa slučajnim pristupom. Imperativni programski jezici, poput Fortrana i Pascala, kao i asemblerski jezici, temelje se na konceptu Turingovog stroja: izvođenju niza instrukcija. S druge strane, funkcionalni programski jezici, poput Miranda-e, ML-a i Lisp-a, temelje se na lambda računu. [3, str. 5]

1.1. Povijest lambda računa

[4, str. 6-8]

Lambda račun izumio je Alonzo Church, koji ga je objavio 1932. godine, iako ga je izumio još oko 1928. godine. Church je rođen 1903. u Washington D.C.-u u Sjedinjenim Američkim Državama, a studirao je na Sveučilištu u Princetonu. Svoju profesionalnu karijeru proveo je u Princetonu do 1967., iako je 1928. i 1929. boravio u Göttingenu i Amsterdamu.

Oko 1928. počeo je razvijati formalni sustav s ciljem davanja temelja za logiku, koji bi bili prirodniji od Russelove teorije tipova i Zermelove teorije skupova, te koji ne bi sadržavali slobodne varijable. Church je odlučio temeljiti svoj sustav, kojeg je nazvao lambda račun (λ-račun), na konceptu funkcija, umjesto na skupovima. Zanimljiva je priča kako se Church odlučio nazvati svoj sustav lambda računom. Prije pojave lamda računa, postojao je zapis "$\hat{x}$", koji se koristio za apstrakciju klasa, a čiji autori su Whitehead i Rusell. Church je najprije nazvao svoj sustav "$\wedge x$", međutim to je promijenjeno u "λ" zbog jednostavnijeg zapisa i čitljivosti.

Slika 1.1: Alonzo Church - preuzeto s www.apprendre-math.info

Church nije prvi uveo zapis za apstrakciju funkcija, ali je prvi izrazio formalna pravila unutar sustava. Prije njega su na apstrakciji funkcija radili Peano i Frege, a njihov rad Churchu nije bio poznat u vremenu razvijanja vlastitog sustava. Churchov sustav iz 1932. godine je uključivao logiku neovisnu o tipovima s neograničenom kvantifikacijom, ali bez zakona isključenja trećeg. Sustav je imao uključena formalna pravila za λ-konverziju. Gotovo odmah nakon objavljivanja, pronađena je kontradikcija unutar sustava. Sustav je zbog toga ispravljen 1933. godine.

Od 1931. do 1934. godine Church je na sveučilištu u Princetonu imao pomoć dvojice studenata, Stephena Kleene-a i Barkleya Rossera. Church, Kleene i Rosser su u četiri godine suradnje došli do velikog broja otkrića vezanih uz Churchovu logiku i "λ"-račun. Nažalost, jedno od tih otkrića je bila nekonzistentnost logike, budući da je dokazana jedna vrsta Richardovog paradoksa unutar sustava. S druge strane, za "λ"-račun, za kojeg se ispočetka činilo da je plitak, pokazao se vrlo bogatim i konzistentnim. Lambda račun danas ima primjenu u računalnoj znanosti, matematici, filozofiji, pa čak i lingvistici.

1.2. Redukcija i funkcionalno programiranje

[3, str. 6]

Funkcionalni program se sastoji od izraza E, koji predstavlja i algoritam i ulaz. Izraz E se transformira prema određenim pravilima. Redukcija je zamjena dijela P od E s nekim drugim izrazom P', prema zadanim pravilima. To možemo zapisati kao:

$$E[P] \to E[P'],$$

pod uvjetom da je $P \to P'$ u skladu s pravilima. Ovaj proces redukcije se ponavlja tako dugo dok više nije moguće primjeniti pravila na dobiveni rezultat. Takozvana normalna forma E* izraza E se sastoji od izlaza danog funkcionalnog programa.

Primjer:

$$\begin{aligned}(5+6)\cdot(2+9\cdot 3) &\to 11\cdot(2+9\cdot 3)\\ &\to 11\cdot(2+27)\\ &\to 11\cdot 29\\ &\to 319.\end{aligned}$$

U ovom primjeru redukcijska pravila sastoje se od "tablica" zbrajanja i množenja prirodnih brojeva. Pomoću redukcije moguće je izvršavati i simboličke izračune. Primjerice:

$$\begin{aligned}&prvi\,od(sortiraj(pridodaj('kivi',' jabuka')(sortiraj(('banana',' ananas'))))))\\ &\quad\to prvi\,od(sortiraj(pridodaj('kivi',' jabuka')('ananas',' banana')))\\ &\quad\to prvi\,od(sortiraj('kivi',' jabuka',' ananas',' banana'))\\ &\quad\to prvi\,od('ananas',' banana',' jabuka',' kivi')\\ &\quad\to\ 'ananas'.\end{aligned}$$

Redukcijski sustavi obično zadovoljavaju Church-Rosserovo svojstvo, koje govori da je dobivena normalna forma nezavisna od redoslijeda evaluacije podtermova. Tako se primjerice gornji primjer može reducirati na sljedeći način:

$$\begin{aligned}(5+6)\cdot(2+9\cdot 3) &\to (5+6)\cdot(2+27)\\ &\to 11\cdot(2+27)\\ &\to 11\cdot 29\\ &\to 319,\end{aligned}$$

ili čak tako da se istovremeno obrađuje nekoliko izraza:

$$\begin{aligned}(5+6)\cdot(2+9\cdot 3) &\to 11\cdot(2+27)\\ &\to 11\cdot 29\\ &\to 319.\end{aligned}$$

2. Lambda račun

[3, str. 7-13]

2.1. Aplikacija i apstrakcija

Prva osnovna operacija lambda računa je aplikacija. Izraz

$$F \cdot A$$

ili

$$FA$$

označava da se podatak F, odnosno algoritam, primjenjuje na podatak A, odnosno ulaz.

Lambda račun je neovisan o tipovima: dozvoljeno je razmatrati izraze poput FF. Ovaj primjer (FF) govori da se algoritam F primjenjuje na samog sebe. Činjenica da je lambda račun neovisan o tipovima je korisna za simuliranje rekurzije.

Druga osnovna operacija lambda računa je apstrakcija. Ako je $M \equiv M[x]$ izraz koji sadrži (zavisi o) x, tada $\lambda x.M[x]$ označava funkciju $x \mapsto M[x]$. Aplikacija i apstrakcija su kombinirane u sljedećem primjeru:

$$(\lambda x.4 \cdot x + 1)7 = 4 \cdot 7 + 1 = 28 + 1 = 29.$$

Izraz $(\lambda x.4 \cdot x + 1)7$ izražava funkciju $x \mapsto 4 \cdot x + 1$. Ako taj izraz primjenimo na argument 7, dobivamo vrijednost $4 * 7 + 1$, što je u konačnici vrijednost 29. Općenito, gornji izraz je oblika $(\lambda x.M[x])N = M[N]$. Ovu jednadžbu možemo zapisati kao

$$(\lambda x.M)N = M[x := N],$$

gdje $[x := N]$ označava zamjenu (supstituciju) N s x. Gornja formula je zapravo jedini esencijalni aksiom lambda računa.

2.2. Slobodne i vezane varijable

Apstrakcija kod lambda računa veže slobodnu varijablu x u M. Primjerice, ako uzmemo izraz $\lambda x.yx$, tada je x vezana varijabla, a y je slobodna varijabla. Supstitucija $[x := N]$ se provodi samo za slobodna pojavljivanja varijable x:

$$yx(\lambda x.x)[x := N] \equiv yN(\lambda x.x).$$

Općenito se pretpostavlja da su vezane varijabe koje se pojavljuju u nekom izrazu različite od slobodnih varijabli. Ta pretpostavka se može ispuniti preimenovanjem vezanih varijabli u izrazima. Primjerice, $\lambda x.x$ možemo preimenovati u $\lambda y.y$. Ta dva izraza imaju isto značenje:

$$(\lambda x.x)a = a = (\lambda y.y)a$$

i zapravo označavaju isti ciljni algoritam.

2.3. Funkcije s više argumenata

Funkcije s nekoliko argumenata moguće je računati iteriranjem aplikacije. Na tu ideju je došao Schönfinkel 1924. godine, ali metoda se često naziva Curryjeva metoda (eng. currying), prema H.B. Curryju, koji ju je uveo kao nezavisnu metodu. Intuitivno, ako $f(x, y)$ zavisi o dva argumenta, moguće je definirati:

$$F_x = \lambda y.f(x, y),$$
$$F = \lambda x.F_x.$$

Tada vrijedi:

$$(Fx)y = F_x y = f(x, y). \tag{2.1}$$

Ova zadnja jednadžba pokazuje da je prikladno koristiti asocijaciju ulijevo za iteriranu lambda aplikaciju:

$FM_1 \cdots M_n$ označava $(\cdot\,(\cdots((FM_1)M_2)\cdots M_n)$.

Jednadžba (2.1) tada postaje:

$$Fxy = f(x, y).$$

Isto tako, iterirana apstrakcija koristi asocijaciju udesno:

$\lambda x_1 \cdots x_n.f(x_1, ..., x_n)$ označava $\lambda x_1.(\lambda x_2.(\cdots(\lambda x_n.f(x_1, ..., x_n))\cdots))$.

Za prethodno definiran F vrijedi:

$$F = \lambda xy.f(x, y),$$

te (2.1) izgleda ovako:

$$(\lambda xy.f(x, y))xy = f(x, y).$$

Općenito, za n argumenata vrijedi sljedeće:

$$(\lambda x_1 \cdots x_n.f(x_1, \cdots, x_n))x_1 \cdots x_n = f(x_1, ..., x_n).$$

2.4. Lambda konverzija

2.4.1. Definicija skupa Λ

Skup λ termova, kojeg označujemo s Λ, sastoji se od neograničenog broja varijabli $V = \{v, v', v'', ...\}$, koje koriste aplikaciju i apstrakciju (funkcija).

$$x \in V \rightarrow x \in \Lambda,$$
$$M, N \in \Lambda \rightarrow (MN) \in \Lambda,$$
$$M \in \Lambda, x \in V \rightarrow (\lambda x M) \in \Lambda.$$

2.4.1.1. Primjeri lambda termova

U sljedećem primjeru nalaze se lambda termovi:

$$v';$$
$$(v'v);$$
$$(\lambda v(v'v));$$
$$((\lambda v(v'v))v'');$$
$$(((\lambda v(\lambda v(v'v)))v'')v''').$$

2.4.2. Konvencija o označavanju varijabli i termova

Prema dogovoru, $x, y, z, ...$ označavaju proizvoljne varijable; $M, N, L, ...$ označavaju proizvoljne λ-termove. Vanjske okrugle zagrade se ne pišu.

$M \equiv N$ označava da su M i N isti term, ili ih je moguće međusobno dohvatiti preimenovanjem vezanih varijabli.

2.4.2.1. Primjeri ekvivalencije termova

Za sljedeće termove možemo reći da su ekvivalentni:

$$(\lambda xy)z \equiv (\lambda yx)z;$$
$$(\lambda xx)z \equiv (\lambda yy)z;$$

S druge strane, sljedeći termovi nisu ekvivalentni:

$$(\lambda xx)z \not\equiv z;$$
$$(\lambda xx)z \not\equiv (\lambda xy)z.$$

2.4.3. Kratice kod λ izraza

Koristimo kratice

$$FM_1 \cdots M_n \equiv (\cdot\cdot((FM_1)M_2)\cdots M_n)$$

i

$$(\lambda x_1 \cdots x_n.f(x_1, \cdots, x_n))x_1 \cdots x_n \equiv f(x_1, ..., x_n),$$

pa termove iz primjera 2.4.1.1 možemo zapisati ovako:

$$y;$$
$$yx;$$
$$\lambda x.yx;$$
$$(\lambda x.yx)z;$$
$$(\lambda xy.yx)zw.$$

Obratimo pozornost kako je $\lambda x.yx$ zapravo $(\lambda x(yx))$, a ne $((\lambda x.y)x)$.

2.4.4. Skup slobodnih varijabli

Skup slobodnih varijabli od M, kojeg zapisujemo kao $FV(M)$, definiramo ovako:

$$FV(x) = \{x\};$$
$$FV(MN) = FV(M) \cup FV(N);$$
$$FV(\lambda x.M) = FV(M) - \{x\}.$$

Varijabla M je vezana ako nije slobodna. Primjećujemo kako općenito vrijedi da je varijabla vezana ako se pojavljuje u dijelu izraza koji sadrži "λ".

M je zatvoreni λ term (ili kombinator), ako je $FV(M) = \emptyset$. Skup zatvorenih lambda termova označavamo s Λ°.

2.4.4.1. Supstitucija terma sa slobodnim varijablama drugog terma

Rezultat supstitucije terma N sa slobodnim pojavljivanjima varijable x u termu M, što zapisujemo kao $M[x := N]$, definiramo ovako:

$$\begin{aligned} x[x := N] &\equiv N; \\ y[x := N] &\equiv y, \text{ ako } x \not\equiv y; \\ (M_1 M_2)[x := N] &\equiv (M_1[x := N])(M_2[x := N]); \\ (\lambda y.M_1)[x := N] &\equiv \lambda y.(M_1[x := N]). \end{aligned}$$

2.4.4.2. Primjer slobodnih i vezanih varijabli

Razmotrimo sljedeći λ term:

$$\lambda xy.xyz.$$

Varijable x i y su vezane varijable, a z je slobodna varijabla. Ako preimenujemo varijablu z u y, tada je term $\lambda xy.xyy$ zatvoren.

2.4.5. Konvencija o varijablama

Ako se termovi $M_1, ..., M_n$ pojavljuju u određenom matematičkom kontekstu, primjerice u definiciji ili dokazu, tada se u tim termovima sve vezane varijable razlikuju od slobodnih.

Tako u četvrtoj klauzuli primjera 2.4.4.1 nije potrebno specificirati izraz "pod uvjetom da $y \not\equiv x$ i $y \notin FV(N)$". Prema konvenciji o varijablama, to se podrazumijeva.

2.4.6. β-redukcija

Osnovna shema aksioma λ-računa glasi:

$$(\lambda x.M)N = M[x := N]$$

za sve $M, N \in \Lambda$. Ova formula naziva se β-redukcija.

Postoje i "logički" aksiomi i pravila. Aksiomi jednakosti:

$$\begin{aligned} M &= M; \\ M = N &\Rightarrow N = M; \\ M = N, N = L &\Rightarrow M = L. \end{aligned}$$

Pravila kompatibilnosti:

$$M = M' \Rightarrow MZ = M'Z;$$
$$M = M' \Rightarrow ZM = ZM';$$
$$M = M' \Rightarrow \lambda x.M = \lambda x.M'.$$

Ako je moguće dokazati $M = N$ u lambda računu, to ponekad zapisujemo u obliku $\boldsymbol{\lambda} \vdash M = N$.

Zahvaljujući pravilima kompatibilnosti, moguće je zamijeniti termove ili podtermove jednakim termovima ili podtermovima u bilo kojem kontekstu termova:

$$M = N \rightarrow \cdots M \cdots = \cdots N \cdots.$$

Primjerice, $(\lambda y.yy)x = xx$, pa vrijedi:

$$\boldsymbol{\lambda} \vdash \lambda x.x((\lambda y.yy)x)x = \lambda x.x(xx)x.$$

2.4.7. α-konverzija

Pravilo α-konverzije glasi:

$$\lambda x.M = \lambda y.M[x := y], \text{pod uvjetom da se } y \text{ ne pojavljuje u } M.$$

2.4.8. Primjer standardnih kombinatora

Definirajmo kombinatore:

$$\boldsymbol{I} \equiv \lambda x.x;$$
$$\boldsymbol{K} \equiv \lambda xy.x;$$
$$\boldsymbol{K_*} \equiv \lambda xy.y;$$
$$\boldsymbol{S} \equiv \lambda xyz.xz(yz).$$

Pomoću indukcije i β-redukcije moguće je dokazati sljedeće:

$$\boldsymbol{\lambda} \vdash (\lambda x_1 \cdots x_n.M)X_1 \cdots X_n = M[x_1 := X_1] \cdots [x_n := X_n].$$

Koristeći taj izraz, možemo izračunati sljedeće:

$$\boldsymbol{I}M = M;$$
$$\boldsymbol{K}MN = M;$$
$$\boldsymbol{K_*}MN = N;$$
$$\boldsymbol{S}MNL = ML(NL).$$

2.4.9. Teorem fiksne točke

Teorem fiksne točke glasi:

$$\forall F\ \exists X\ FX = X, \tag{2.2}$$

što znači da za svaki $F \in \Lambda$ postoji neki $X \in \Lambda$ takav da je $\boldsymbol{\lambda} \vdash FX = X$. Postoji i kombinator fiksne točke:

$$\boldsymbol{Y} \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

takav da

$$\forall F\ F(\boldsymbol{Y}F) = \boldsymbol{Y}F. \tag{2.3}$$

Jednadžbu (2.2) dokazujemo tako da definiramo $W \equiv \lambda x.F(xx)$ i $X \equiv WW$. Tada vrijedi:

$$X \equiv WW \equiv (\lambda x.F(xx))W \equiv F(WW) \equiv FX.$$

Jednadžbu (2.3) dokazujemo prema dokazu jednadžbe (2.2).

2.4.9.1. Primjer fiksne točke

Dokažimo da vrijedi: $\exists G\ \forall X\ GX = \boldsymbol{S}GX$.

$$\begin{aligned} \forall X\ GX = SGX &\Leftarrow Gx = \boldsymbol{S}Gx \\ &\Leftarrow G = \lambda x.\boldsymbol{S}Gx \\ &\Leftarrow Gx = (\lambda gx.\boldsymbol{S}gx)G \\ &\Leftarrow Gx = \boldsymbol{Y}(\lambda gx.\boldsymbol{S}gx) \end{aligned}$$

Primjećujemo kako vrijedi i $G \equiv \boldsymbol{YS}$.

2.4.10. Churchovi numerali

Kod lambda računa postoji nekoliko načina za definiranje brojeva, a najuobičajeniji način je definiranje brojeva korištenjem Churchovih numerala. Neka je izraz $F^n(M)$, gdje je $F \in \Lambda$ i $n \in \mathbb{N}$, definiran kao što slijedi:

$$F^0(M) \equiv M;$$
$$F^{n+1}(M) \equiv F(F^n(M)).$$

Churchove numerale $c_0, c_1, c_2, ...$ definiramo na sljedeći način:

$$c_n \equiv \lambda fx.f^n(x).$$

U sljedećem poglavlju bit će opisana primjena lambda izraza u programskom jeziku Python.

3. Lambda račun i Python

[1]

3.1. Funkcionalno programiranje u Pythonu

Prije pojave objektno-orijentiranog programiranja i programskog jezika Python, jedna od glavnih grana računalnog programiranja bilo je funkcionalno programiranje. Za računalno programiranje, funkcija je račun i može se definirati kao skup rezultata koji su dobiveni izvođenjem niza predvidivih operacija nad ulaznim vrijednostima. U programiranju, te parametre nazivamo argumentima funkcije.

Dobar način za zapis matematičke funkcije u pseudokodu je zapisivanje korištenjem lambda računa. Primjerice, za kvadrat nekog broja x, matematička formula glasi $x \cdot x$. Međutim, kod lambda računa pišemo formalnu definiciju funkcije:

$$\lambda x.x \cdot x$$

Navedena funkcija, čiji argument se naziva "x", vraća vrijednost "x*x".

Funkcionalni programski jezici, poput Lisp-a, se temelje na lambda računu. Programski jezik Python omogućuje funkcionalno programiranje, a podržava i korištenje lambda računa. Pythonova "lambda" funkcionalnost je zapravo alat za stvaranje funkcija, odnosno funkcijskih objekata. To znači da Python ima dva alata za stvaranje funkcija: "def" i "lambda". Gore navedenu funkciju u Pythonu možemo koristiti na standardni način, pomoću konstrukta "def":

```
>>> def kvadratDef(x):
...     return x*x
```

U Pythonu je moguće korištenje "lambda" konstrukata i za definiranje anonimnih funkcija (funkcija koje nisu vezane uz naziv). Radi se o vrlo snažnom konceptu koji je dobro integriran u programski jezik Python, a njegova moć se očituje prilikom kombiniranja s tipčinim funkcijskim konceptima poput "filter()", "map()" i "reduce()" koncepata.

Poput "def" konstrukta, "lambda" konstrukt stvara funkciju koju možemo pozivati. Međutim, "lambda" konstrukt vraća funkciju, dok "def" konstrukt veže funkciju uz naziv. Iz tog razloga lambda funkcije ponekad nazivamo anonimnim funkcijama. U praksi se lambda izrazi najčešće koriste za ugrađivanje definicije funkcije.

3.2. Lambda funkcije s jednim argumentom

Funkciju iz prethodnog primjera u Pythonu možemo definirati korištenjem "lambda" konstrukta:

```
>>> kvadrat = lambda x: x*x
```

Primjećujemo kako "lambda" definicija funkcije ne sadrži izjavu "return", već sadrži izraz kojeg vraća. Lambda definiciju možemo staviti na svakom mjestu gdje očekujemo funkciju, te joj ne trebamo dodijeliti nikakav naziv. Isto tako, vidimo kako je sintaksa matematičkog zapisa lambda funkcije i zapisa lambda funkcije u Pythonu vrlo slična. Lambda funkciju možemo pozvati na sljedeći način:

```
>>> kvadrat(5)
25
```

Python interpreter kao parametar uzima vrijednost broja, te izračunava vrijednost funkcije. U gornjem primjeru, za parametar 5, vrijednost funkcije iznosi 25.

Definirajmo sada funkciju "drugiKorijen":

```
>>> import math
>>> drugiKorijen = lambda x: math.sqrt(x)
>>> drugiKorijen(121)
11.0
```

Za funkciju drugog korijena ("sqrt"), u Pythonu je potrebno uvesti modul "math".

Na isti način možemo i definirati funkcije inkrementa, koji povećava vrijednost parametra za 1, i dekrementa, koji smanjuje vrijednost parametra za 1:

```
>>> inkrement = lambda x: x+1
>>> dekrement = lambda x: x-1
```

Možemo testirati definirane lambda funkcije:

```
>>> inkrement(2)
3
>>> dekrement(6)
5
```

Vidimo kako funkcije vraćaju ispravne vrijednosti, pa možemo reći da su dobro definirane.

Python automatski prepoznaje tip podatka parametra, pa je tako moguće kombinirati lambda funkcije s različitim tipovima podataka. U sljedećem primjeru pozivaju se lambda funkcije inkrementa i kvadrata za decimalne vrijednosti:

```
>>> inkrement(1.1)
2.1
>>> kvadrat(2.2)
4.84
```

Nakon što definiramo funkcije, možemo ih pozivati kao u sljedećem primjeru:

```
>>> inkrement(inkrement(1))
3
>>> inkrement(inkrement(inkrement(1)))
4
>>> dekrement(dekrement(5))
3
>>> inkrement(dekrement(5))
5
```

Općenito u programiranju vrijedi da se prvo pozivaju unutarnje funkcije. Nakon što funkcija vrati određenu vrijednost, poziva se vanjska funkcija, koja kao parametar uzima izračunatu vrijednost unutarnje funkcije. Tako se u gornjem primjeru u posljednjem pozivu funkcija, "inkrement(dekrement(5))", prvo poziva "dekrement(5)", koja vraća vrijednost 4. Potom funkcija "inkrement" prima parametar 4 i vraća vrijednost 5.

3.3. Lambda funkcije s više argumenta

Za funkcije s više argumenata koristimo Curryjevu metodu, koja je opisana u poglavlju 2.3.

3.3.1. Funkcije s dva argumenta

Uzmimo za primjer funkciju zbroja dvaju pribrojnika. Matematički zapis te funkcije izgleda ovako:

$$\lambda xy.x + y.$$

Istu funkciju u Pythonu možemo definirati na sljedeći način:

```
>>> zbroj = lambda x: lambda y: x+y
```

Vidimo kako funkcija ima dva argumenta, pa je stoga prilikom pozivanja funkcije potrebno unijeti dva parametra:

```
>>> zbroj(13)(37)
50
>>> zbroj(25)(-40)
-15
>>> zbroj(13.37)(1337)
1350.37
```

U nastavku slijedi primjer funkcije koja za unesena dva elementa vraća vrijednost manjeg, i to pomoću Python funkcije "min":

```
>>> minDvaBroja = lambda x, y: min(x, y)
>>> minDvaBroja(13,37)
13
>>> minDvaBroja(37,13)
13
```

U Pythonu je unutar lambda izraza moguće korištenje svih standardnih Python funkcija, kao i korisnički definiranih funkcija.

3.3.2. Funkcije s tri ili više argumenata

U prethodnom primjeru funkciju zbroja definirali smo na sljedeći način:

```
>>> zbroj = lambda x: lambda y: x + y
```

Međutim, ako imamo više argumenata, moguće ih je odvojiti zarezom unutar istog izraza "lambda":

```
>>> zbroj = lambda x, y: x+y
```

Ovakav pristup (korištenje asocijacije udesno) nam ubrzava pisanje i poboljšava čitljivost koda. Uzmimo za primjer funkciju zbroja 5 pribrojnika:

```
>>> zbrojPetPribrojnika = lambda x1: lambda x2: lambda x3: lambda x4:
   lambda x5: x1 + x2 + x3 + x4 + x5
>>> zbrojPetPribrojnika(1)(2)(3)(4)(5)
15
```

Korištenjem asocijacije udesno možemo skratiti definiciju iste funkcije:

```
>>> zbrojPetPribrojnika = lambda x1, x2, x3, x4, x5: x1 + x2 + x3 + x4
   +x5
```

Tako definiranu funkciju pozivamo unošenjem parametara koje odvajamo zarezom, što je također brži način:

```
>>> zbrojPetPribrojnika(1,2,3,4,5)
15
```

Funkciju zbroja pet pribrojnika možemo iskoristiti za definiciju prosjeka pet brojeva:

```
>>> prosjekPetBrojeva = lambda x1, x2, x3, x4, x5:
   zbrojPetPribrojnika(x1, x2, x3, x4, x5)/5
>>> prosjekPetBrojeva(1, 2, 3, 4, 5)
3
```

3.4. Primjer kombinacije lambda izraza s "def" konstruktom

Lambda izraze moguće je kombinirati i s "def" konstruktom, pomoću koje se definiraju funkcije. U nastavku je primjer funkcije inkrementa za vrijednost n:

```
>>> def inkrementZaN(n):
...   return lambda x: x+n
...
>>> inkrementZa10 = inkrementZaN(10)
>>> inkrementZa10(60)
70
```

Iako je moguće korištenje lambda izraza unutar definicije funkcije pomoću konstrukta "def", funkcija inkrementa za vrijednost n može se jednostavnije zapisati na sljedeći način:

```
>>> inkrementZaN = lambda x: lambda y: x+y
>>> inkrementZa10 = inkrementZaN(10)
>>> inkrementZa10(60)
70
```

Međutim, iako funkcijski objekti dobiveni pomoću lambda izraza rade potpuno isto kao i oni kreirani pomoću "def" konstrukta, ipak postoje određene razlike:

- Lambda je izraz, a ne izjava.
  Zbog tog svojstva, lambda izraz se može pojaviti na mjestima gdje "def" konstrukt nije dozvoljen, na primjer unutar literala liste ili argumenata poziva funkcije. Kao izraz, lambda vraća vrijednost kojoj opcionalno možemo dodijeliti naziv. S druge strane, "def" izjava uvijek funkciji dodjeljuje naziv.

- Tijelo lambda izraza je jedan izraz, a ne blok izjava.
  Tijelo lambda izraza je slično izrazu koji bismo umetnuli u "return" izjavu "def" konstrukta. Budući da je limitirana na izraz, lambda izjava je manje općenita od "def" izraza. Lambda izrazi su prije svega namijenjeni kodiranju jednostavnih funkcija, dok su "def" izrazi prikladniji za zahtjevnije zadatke.

3.5. Python i α-konverzija

Postavlja se pitanje hoće li lambda funkcija funkcionirati ako promijenimo nazive varijabli. Zamijenimo varijablu x s varijablom z unutar funkcije zbroja:

```
>>> zbroj = lambda z: lambda y: z+y
>>> zbroj(13)(37)
50
```

Primjećujemo kako funkcija "zbroj" vraća ispravnu vrijednost. Mogućnost promjene naziva vezanih varijabli je jedno od glavnih pravila lambda računa, te je opisano u poglavlju 2.4.7 (α-konverzija).

Na sličan način možemo definirati čitav niz matematičkih funkcija:

```
>>> razlika = lambda x : lambda y: x-y
>>> razlika(13)(37)
-24
>>>
>>> umnozak = lambda x: lambda y: x*y
>>> umnozak(13)(37)
481
>>>
>>> kvocijent = lambda djeljenik: lambda djelitelj: djeljenik/djelitelj
>>> kvocijent(100)(5)
20
>>>
>>> ostatak = lambda x: lambda y: x%y
>>> ostatak(37)(13)
11
>>>
>>> potencija = lambda osnova: lambda eksponent: pow(osnova, eksponent)
>>> potencija(2)(4)
16
>>>
>>> import math
>>> drugiKorijenZbroja = lambda x: lambda y: math.sqrt(x + y)
>>> drugiKorijenZbroja(12+4)
4.0
```

3.6. Python i β-redukcija

U programiranju općenito vrijedi da funkcija koja ima određene argumente mijenja sve reference na taj argument sa zadanim vrijednostima, te vraća rezultat. To nazivamo aplikacijom (primjenom) funkcije. Pogledajmo sada primjer funkcije zbroja:

```
>>> zbroj = lambda x: lambda y: x+y
>>> zbroj13 = zbroj(13)
#Zamijenili smo varijablu x s vrijednošću 13.
#zbroj13 = lambda 13: lambda y: 13+y
#zbroj13 = lambda y: 13+y

>>> zbroj13(37)
#Primjenjujemo aplikaciju funkcije
#Lambda 37: 13+37
50
```

Kod lambda računa, ovakva primjena aplikacije funkcije naziva se β-redukcija, koja je formalno opisana u poglavlju 2.4.6. Gornji primjer u sintaksi lambda računa izgleda ovako:

$$(\lambda xy.x + y)13$$
$$(\lambda y.13 + y)37$$
$$13 + 37$$
$$50$$

3.7. Python i η-konverzija

Računalne funkcije mogu raditi istu stvar, a pritom imati vrlo različite implementacije. Primjerice, funkcija:

```
>>> dvostrukaVrijednost1 = lambda x: x+x
>>> dvostrukaVrijednost1(120)
240
```

... daje potpuno isti rezultat za sve vrijednosti kao i sljedeća funkcija:

```
>>> dvostrukaVrijednost2 = lambda x: x*2
>>> dvostrukaVrijednost2(120)
240
```

U dijelu koda koji koristi funkciju "dvostrukaVrijednost1" možemo lako zamijeniti navedenu funkciju s funkcijom "dvostrukaVrijednost2", a program će i dalje funkcionirati na isti način. Kod lambda računa, ova funkcionalnost naziva se η-konverzija.

4. Lambda račun i nenumerički tipovi podataka

[1]

Lambda račun u Pythonu možemo kombinirati s velikim brojem tipova podataka i objekata: listama, rječnicima, brojevima, znakovima, tekstom, korisnički definiranim objektima, i tako dalje.

4.1. Lambda račun i tekst

Lambda račun možemo koristiti za operacije nad tekstom. Pritom koristimo standardne funkcije iz Pythona u kombinaciji s lambda izrazom. Pogledajmo primjer spajanja dviju riječi:

```
>>> spojiDvijeRijeci = lambda x,y: x + ' ' + y
>>> spojiDvijeRijeci('Lambda','racun')
'Lambda racun'
```

Primjer prebrojavanja pojavljivanja znaka u zadanom tekstu:

```
>>> prebrojiPojavljivanjeZnaka = lambda tekst, znak: tekst.count(znak)
>>> prebrojiPojavljivanjeZnaka('Neka sila bude s tobom', 'a')
2
```

Primjer funkcije koja mijenja znakovni niz unutar teksta:

```
>>> zamjenaTeksta = lambda tekst, stariTekst, noviTekst:
   tekst.replace(stariTekst, noviTekst)
>>> zamjenaTeksta('Ovo je poslastica za sve kosarkaske sladokusce.',
   'sladokusce', 'fanove')
'Ovo je poslastica za sve kosarkaske fanove.'
```

U nastavku se nalazi primjer u kojem se nalazi još jedan način definiranja vezanih varijabli:

```
>>> magistarStruke = (lambda a = 'Pero', b = ' Peric', c = ',
   mag.inf.': a + b + c)
>>> magistarStruke()
'Pero Peric, mag.inf.'
>>>
>>> magistarStruke('Ivo', ' Ivic')
'Ivo Ivic, mag.inf.'
>>>
```

U gornjem primjeru vezanim varijablama dodjeljujemo vrijednosti prilikom njihove definicije. Pozivom funkcije bez argumenata poziva se lambda funkcija s definiranim vrijednostima. Pozivom funkcije s dva argumenta mijenjaju se prva dva argumenta.

Python omogućuje korištenje varijabli unutar lambda izraza. Napravimo funkciju s atributom "naslov", koja će unesenom tekstu dodati prefiks:

```
>>> def prefiks():
...     naslov = 'Cijenjeni i uvazeni gospodin'
...     ime = (lambda x: naslov + ' ' + x)
...     return ime
>>>
>>> pre = prefiks()
>>> pre('Ivo Ivic')
'Cijenjeni i uvazeni gospodin Ivo Ivic'
```

U gornjem smo primjeru unutar lambda izraza koristili atribut "naslov" unutar funkcije "prefiks()".

4.2. Lambda račun i liste

U Pythonu je moguće koristiti lambda izraze i za liste. Definirajmo listu koja sadrži lambda izraze pomoću kojih izračunavamo kvadrat, kub i četvrtu potenciju zadanog broja:

```
>>> listaPotencija = [lambda x: x**2,
...                   lambda x: x**3,
...                   lambda x: x**4]
```

Elementima liste u Pythonu pristupamo pomoću "for ... in" konstrukta:

```
>>> for element in listaPotencija:
...   element(4)
...
16
64
256
```

Budući da se radi o listi, možemo pristupiti svakom elementu liste pomoću indeksa. Izračunajmo kvadrat broja 17. Lambda izraz koji računa kvadrat nalazi se na nultom indeksu liste:

```
>>> kvadrat = listaPotencija[0]
>>> kvadrat(17)
289
```

U ovom primjeru inicijalizirali smo funkciju "kvadrat" tako što smo dohvatili lambda izraz koji se nalazi na nultom indeksu liste. Lambda izraz unutar liste moguće je dohvatiti i bez inicijalizacije funkcije:

```
>>> listaPotencija[0](17)
289
```

U gornjem primjeru napravljena je lista triju funkcija korištenjem lambda izraza unutar liste. To nije moguće napraviti korištenjem "def" konstrukta, budući da je "def" izjava, a ne izraz. Ako želimo implementirati iste funkcije pomoću "def" konstrukta, moramo koristiti privremene nazive funkcija i pripadajuće definicije:

```
>>> def kvadrat(x): return x**2
...
>>> def kub(x): return x**3
...
>>> def cetvrtaPotencija(x): return x**4
...
>>> #Referenciranje prema nazivu
>>> listaPotencija = [kvadrat, kub, cetvrtaPotencija]
>>> for element in listaPotencija:
...   element(4)
...
16
64
256
>>> listaPotencija[0](17)
289
```

Usporedimo li gornja dva primjera, jasno je vidljivo kako je korištenje lambda izraza brži i jednostavniji način, a uz to nije potrebno korištenje naziva funkcija, kao ni referenciranje na te nazive.

4.3. Lambda račun i rječnici

Na sličan način kao što smo definirali liste, možemo definirati i rječnike koji se sastoje od lambda izraza:

```
>>> rjecnikPotencija = {'kvadrat':(lambda x: x**2), 'kub':(lambda x:
   x**3), 'cetvrtaPotencija':(lambda x: x**4)}
>>> rjecnikPotencija['kvadrat'](17)
289
```

Ako želimo implementirati takav rječnik pomoću "def" konstrukta, kod izgleda ovako:

```
>>> def kvadrat(x): return x**2
...
>>> def kub(x): return x**3
...
>>> def cetvrtaPotencija(x): return x**4
...
>>> rjecnikPotencija = {'kvadrat': kvadrat, 'kub': kub,
   'cetvrtaPotencija': cetvrtaPotencija}
>>>
>>> rjecnikPotencija['kvadrat'](17)
289
```

Ponovno je jasno vidljivo kako pomoću lambda izraza znatno skraćujemo i pojednostavljujemo kod. Osim toga, "def" funkcije koje se pozivaju unutar rječnika se u kodu mogu nalaziti daleko od samog rječnika. U takvim situacijama je još korisnije koristiti lambda izraze, a posebice ako se funkcije neće koristiti nigdje drugdje. Isto tako, "def" funkcije se vežu uz naziv, pa se povećava vjerojatnost preklapanja naziva s nekim drugim objektom unutar koda.

4.4. Kontrolne petlje u lambda računu

[8]

U Pythonu je moguće korištenje kontrolnih petlji unutar lambda izraza.

4.4.1. If-else petlja

"If-else" petlja ima sljedeću sintaksu:

```
'rezultat1' if 'uvjet' else 'rezultat2'.
```

Prema tome, ako je uvjet zadovoljen, vraća se vrijednost "rezultat1", a ako uvjet nije zadovoljen, vraća se vrijednost "rezultat2".

Implementirajmo funkciju koja će za uneseni broj ispisati je li broj manji od broja 100:

```
>>> jeManjiOd100 = lambda x: 'je manji od 100' if x<100 else 'nije
    manji od 100'
```

Navedena funkcija ispisuje poruku "manji od 100" u slučaju da je vrijednost parametra "x" manja od 100. U slučaju da vrijednost parametra "x" nije manja od 100, ispisuje se poruka "nije manji od 100". Možemo ispitati funkciju:

```
>>> jeManjiOd100(1)
'je manji od 100'
>>> jeManjiOd100(99)
'je manji od 100'
>>> jeManjiOd100(100)
'nije manji od 100'
>>> jeManjiOd100(1337)
'nije manji od 100'
```

Kao što je već spomenuto, lambda funkcije su naročito korisne kada trebamo funkciju koju ćemo pozivati samo na jednom mjestu u programu. Primjerice, recimo da imamo listu brojeva koja je zadana na sljedeći način:

```
>>> listaBrojeva = [1, 70, 93, 120, 201]
```

U slučaju da je potrebno provjeriti koji su brojevi unutar liste parni, a znamo da to vjerojatno nećemo koristiti ni na jednom drugom mjestu, tada lambda funkcija dolazi do punog izražaja:

```
>>> for broj in listaBrojeva:
...    print broj, ' ', (lambda x: 'je paran' if x%2 is 0 else 'nije
   paran')(broj)
...
1  nije paran
70  je paran
93  nije paran
120  je paran
201  nije paran
```

Da smo istu funkciju definirali pomoću "def" konstrukta, kod bi bio malo složeniji, koristili bismo naziv funkcije za alokaciju, a definicija funkcije bi se nalazila izvan mjesta pozivanja.

4.4.2. And-or petlja

Postoji i "and-or" kontrolna petlja, koja radi isto što i "if-else" petlja, a funkcionira na sljedeći način:

```
'uvjet' and 'rezultat1' or 'rezultat2'
```

Ako je "uvjet" ispunjen, petlja vraća "rezultat1", a inače vraća "rezutat2". Implementirajmo sada lambda funkciju pomoću "and-or" kontrolne petlje:

```
>>> jeManjiOd10 = lambda x: x < 10 and 'je manji od 10' or 'nije manji
   od 10'
>>> jeManjiOd10(1)
'je manji od 10'
>>> jeManjiOd10(9)
'je manji od 10'
>>> jeManjiOd10(10)
'nije manji od 10'
>>> jeManjiOd10(17)
'nije manji od 10'
```

Pomoću "and-or" petlje moguće je ispitivati više uvjeta, a za to koristimo sljedeću sintaksu:

```
('uvjet1 and 'rezultat1') or ... or ('uvjetn' and 'rezultatn') or
   'rezultatx'
```

Provjeravanje počinje od prvog uvjeta. Ako je uvjet ispunjen, vraća se prvi rezultat. Ako uvjet nije ispunjen, petlja prelazi na sljedeći uvjet i ispituje ga na isti način. Ako nijedan od uvjeta nije ispunjen, moguće je postaviti vrijednost koja se vraća u tom slučaju, a u gornjem primjeru sintakse to je "x".

Implementirajmo sada funkciju koja računa razmak između dva broja:

```
>>> razmakIzmeduDvaBroja = lambda x, y: (x > y and x - y) or (y > x and
    y - x) or (x == y and 0) or 0
>>> razmakIzmeduDvaBroja(13,37)
24
>>> razmakIzmeduDvaBroja(37,13)
24
>>> razmakIzmeduDvaBroja(37,37)
0
```

Funkcija ima dva argumenta: "x" i "y". Ako je prvi argument veći od drugog, drugi se oduzima od prvog, te funkcija vraća dobivenu vrijednost. Isto tako, ako je drugi argument veći od prvog, prvi se oduzima od drugog, te funkcija vraća dobivenu vrijednost. Ako su argumenti jednaki, funkcija vraća vrijednost 0. Postavljena je i vrijednost koja se vraća u slučaju da nijedan od uvjeta nije zadovoljen, ali to u ovom primjeru nije potrebno, budući da su prethodno pokrivene sve mogućnosti.

Ukoliko nam "or" dio petlje nije potreban, jednostavno možemo izostaviti taj dio, pa je sintaksa još jednostavnija:

```
'uvjet' and 'rezutat'
```

Napravimo sada lambda funkciju koja će ispisati vrijednost samo ako je unesen parametar s vrijednošću 17:

```
>>> jeLiBroj17 = lambda x: x is 17 and 'broj je 17'
>>> jeLiBroj17(17)
'broj je 17'
>>> jeLiBroj17(37)
False
```

Ako parametar ima vrijednost 17, ispisuje se odgovarajuća poruka. Kako nema "or" dijela petlje, funkcija za parametre različite od 17 vraća "false".

5. Izrazima orijentirane funkcije u Pythonu

[1]

Osim samih lambda izraza, Python sadrži još nekoliko funkcije koje omogućuju funkcionalni pristup programiranju. Te funkcije također skraćuju kod, a njihovo korištenje je relativno jednostavno.

Funkcionalno programiranje svodi se na pisanje izraza, pa možemo reći da je funkcionalno programiranje zapravo izrazima orijentirano programiranje. Izrazima orijentirane funkcije koje omogućuje Python su:

- map(aFunkcija, aSekvenca)
- filter(aFunkcija, aSekvenca)
- reduce(aFunkcija, aSekvenca)
- lambda
- komprehencija liste

5.1. Map

Jedna od najčešćih operacija koju izvršavamo nad listama i drugim sekvencama (nizovima) je primjenjivanje određene operacije nad svakim elementom, te skupljanje rezultata. Primjerice, izračunavanje novih vrijednosti elemenata liste moguće je lako implementirati pomoću "for" petlje:

```
>>> lista = [1, 5, 7, 13, 37]
>>> kvadrati = []
>>> for element in lista:
...     kvadrati.append(element ** 2)
...
>>> kvadrati
[1, 25, 49, 169, 1369]
```

U gornjem primjeru instancirali smo listu koja se sastoji od 5 cijelih brojeva. Instancirali smo i praznu listu "kvadrati". Svakom elementu pristupili smo pomoću "for" petlje, kvadrirali smo ga, te smo ga pomoću funkcije "append" dodali listi "kvadrati".

Ovakvu operaciju moguće je izvesti i na jednostavniji način, zahvaljujući funkciji "map". Funkcija je oblika "map(aFunkcija, aSekvenca"), primjenjuje funkciju "aFunkcija" na svaki element objekta kojeg je moguće iterirati, odnosno "aSekvenca" objekta, te vraća listu koja sadrži sve rezultate poziva funkcije. Primjenimo sada funkciju "map":

```
>>> lista = [1, 5, 7, 13, 37]
>>> def kvadrat(x): return x ** 2
...
>>> map(kvadrat, lista)
[1, 25, 49, 169, 1369]
```

Instancirali smo novu listu sa istim elementima kao u prethodnom primjeru. Definirali smo funkciju "kvadrat", te smo kao parametre funkcije "map" postavili definiranu funkciju i listu. Na taj način smo pojednostavili kod, budući da nije bilo potrebno pisati "for" petlju.

Kako je prvi argument "map" funkcije zapravo funkcija, vrlo je prikladno na tom mjestu koristiti lambda izraz, pa je moguće dodatno pojednostavniti kod:

```
>>> lista = [1, 5, 7, 13, 37]
>>> map((lambda x: x ** 2), lista)
[1, 25, 49, 169, 1369]
```

Na taj način smo iskoristili lambda funkciju koja kvadrira svaki element liste.

Budući da se radi o funkciji koja je ugrađena u Pyhon, "map" funkcija je uvijek dostupna i uvijek funkcionira na isti način. Isto tako, korištenjem te funkcije program dobiva na performansama, zato jer je funkcija brža od "ručno" pisane "for" petlje. Uz sve to, "map()" je moguće koristiti i na napredniji način. Primjerice, ako "map" prima više argumenata koji su sekvence, tada funkciji prosljeđuje elemente tih sekvenci koji se uparuju prema indeksu:

```
>>> map(pow, [1, 2, 3], [4, 5, 6])
[1, 32, 729]
>>>
```

U gornjem primjeru funkcija map kao parametre prima funkciju "pow", koja je ugrađena u Python, te liste "[1, 2, 3]" i "[4, 5, 6]". Funkciji "pow" redom prosljeđuje elemente "1 i 4", "2 i 5", te "3 i 6". Prema tome, najprije se prosljeđuju elementi s indeksom 0, potom se prosljeđuju elementi s indeksom 1, te tako do posljednjeg elementa sekvence.

Funkcija "map" za funkciju koja ima "N" argumenata očekuje "N" sekvenci. U gornjem primjeru, funkcija "pow" uvijek ima dva argumenta (bazu i eksponent), pa je stoga potrebno korištenje dviju sekvenci.

Ako kao parametar funkcije "map" koristimo "None", pretpostavlja se da se radi o funkciji identiteta: ako postoji više argumenata, "map" funkcija vraća listu koja se sastoji od n-torki koje se sastoje od elemenata sekvenci koji se nalaze na istom indeksu. Neovisno o kojoj vrsti sekvence se radi, funkcija uvijek vraća listu:

```
>>> lista1 = [7, 5, 4]
>>> lista2 = [3, 9, 6]
>>> lista3 = [4, 8, 2]
>>> map(None, lista1, lista2)
[(7, 3), (5, 9), (4, 6)]
>>> map(None, lista1, lista2, lista3)
[(7, 3, 4), (5, 9, 8), (4, 6, 2)]
```

Vidimo da je korištenje "None" parametra u kombinaciji sa funkcijom "map" zapravo slično operaciji transponiranja matrica.

5.2. Filter

Pomoću funkcije "filter" moguće je dohvatiti sve elemente sekvence koji zadovoljavaju određeni uvjet. Možemo reći da funkcija zapravo "filtrira", odnosno uklanja pojedine elemente iz liste. Funkcija "filter" prima dva parametra: funkciju koja se primjenjuje, te sekvencu na koju se primjenjuje ta funkcija. Provjerava se svaki element u sekvence: ako funkcija vrati "true", element ostaje u listi, a inače se izbacuje. Pogledajmo funkciju "filter" na sljedećem primjeru:

```
>>> filter(lambda x: x%2, range(10))
[1, 3, 5, 7, 9]
```

Prvi parametar funkcije "filter" u gornjem primjeru je lambda izraz. Drugi parametar je lista brojeva između 0 i 9, koju je dobivena pomoću funkcije "range". Parametar funkcije "range" je broj 10, i na taj način se vraćaju brojevi od 0 do 9. Broj koji je zadan kao parametar nikad nije uključen u listu koju vraća funkcija "range". Lambda funkcija se primjenjuje na dobivenu listu, te se za svaki element liste ispituje zadovoljava li uvjet "x%2". Element liste zadovoljava navedni uvjet ako pri dijeljenju s brojem 2 vraća ostatak. Konačni rezultat je lista neparnih brojeva između 0 i 9.

Napravimo sada funkciju koja će vratiti listu svih brojeva manjih od 0 u rasponu između -5 i 5:

```
>>> filter(lambda x: x <0, range(-5,5))
[-5, -4, -3, -2, -1]
```

Baš kao i funkcija "map", korištenje funkcije "filter" je slično korištenju "for" petlje na sljedeći način:

```
>>> lista = []
>>> for x in range(-5, 6):
...   if x < 0:
...         lista.append(x)
...
>>> lista
[-5, -4, -3, -2, -1]
```

Jasno je vidljivo da je korištenje funkcije "filter" mnogo brži način od pisanja "for" petlje, a dodatna prednost je brže izvođenje programa, o čemu će biti riječi u idućem poglavlju.

5.3. Reduce

Funkcija "reduce" reducira listu na jednu vrijednost, a također prima dva parametra: funkciju i sekvencu. Funkcija unutar funkcije "reduce" mora imati dva parametra. Unutarnja funkcija (funkcija koja je parametar funkcije "reduce") primjenjuje se na elemente liste, te se vraća izračunata vrijednost. Funkcija "reduce" nalazi se unutar modula "functools".

Pogledajmo funkciju "reduce" na sljedećem primjeru:

```
>>> reduce(lambda x, y: x*y, [1, 2, 3, 4])
24
```

Parametri funkcije iz gornjeg primjera su lambda funkcija i lista. Matematička operacija "x*y" najprije se primjenjuje na prva dva elementa liste: "1" i "2", te se dobiva vrijednost "2". Ista operacija se primjenjuje na izračunatu vrijednost i idući element liste, i taj postupak se ponavlja do zadnjeg elementa liste.

Možemo implementirati funkciju koja ne množi elemente liste, nego ih dijeli:

```
>>> reduce(lambda x, y: x/y, [1024, 2, 4, 8])
16
```

Izvođenje ove funkcije možemo prikazati na sljedeći način:

$$1024 : 2 = 512,$$
$$512 : 4 = 128,$$
$$128 : 8 = 16,$$
$$16$$

Pomoću "for" petlje možemo napraviti vlastiti "reduce" za dijeljenje elemenata:

```
>>> L = [1024, 2, 4, 8]
>>> rezultat = L[0]
>>> for x in L[1:]:
...     rezultat = rezultat / x
...
>>> rezultat
16
```

Ovakav pristup je kompliciraniji i sporiji od korištenja funkcije "reduce".

Možemo i napraviti vlastitu verziju funkcije "reduce" koja prima funkciju i sekvencu kao parametre:

```
>>> def mojReduce(funkcija, sekvenca):
...     rezultat = sekvenca[0]
...     for x in sekvenca[1:]:
...             rezultat = funkcija(rezultat, x)
...     return rezultat
...
>>> mojReduce(lambda x, y: x * y, [1, 2, 3, 4])
24
```

Pomoću funkcije reduce možemo konkatenirati listu koja se sastoji od "string" tipa podataka:

```
>>> lista = ['Dat ', 'cu ', 'mu ', 'ponudu ', 'koju ', 'nece ', 'moci
    ', 'odbiti.']
>>> reduce(lambda x, y: x + y, lista)
'Dat cu mu ponudu koju nece moci odbiti.'
```

Alternativno, za postizanje istog rezultata, možemo koristiti funkciju "add" iz modula "operator", te funkciju "join":

```
>>> ' '.join(lista)
'Dat cu mu ponudu koju nece moci odbiti.'
>>>
>>> import operator
>>> reduce(operator.add, lista)
'Dat cu mu ponudu koju nece moci odbiti.'
```

Funkcija "reduce" može primati i tri parametra, gdje je treći parametar zapravo početna vrijednost. Ako treći parametar nije zadan, tada je početna vrijednost 0.

```
>>> reduce(lambda x,y: x+y, [1,2], 10)
13
```

Da je lista iz gornjeg primjera prazna, tada bi funkcija "reduce" vratila početnu vrijednost:

```
>>> reduce(lambda x,y: x+y, [], 10)
10
```

5.4. Komprehencija liste

U Pythonu, ugrađena funkcija "ord" vraća ASCII vrijednost zadanog znaka:

```
>>> ord('A')
65
```

Pomoću funkcije "map" možemo na jednostavan način dobiti listu ASCII vrijednosti svih znakova u nekom tekstu:

```
>>> map(ord, 'Lambda racun')
[76, 97, 109, 98, 100, 97, 32, 114, 97, 99, 117, 110]
```

Međutim, isti rezultat možemo dobiti ako koristimo komprehenciju liste:

```
>>> [ord(x) for x in 'Lambda racun']
[76, 97, 109, 98, 100, 97, 32, 114, 97, 99, 117, 110]
```

Razlika između funkcije "map" i komprehencije liste je u tome što "map" primjenjuje funkciju nad listom, dok komprehencija liste primjenjuje izraz.

Komprehencija liste je naročito prikladna kada je potrebno primjeniti neki proizvoljni izraz nad listom:

```
>>> [x ** 2 for x in range(5)]
[0, 1, 4, 9, 16]
```

Da smo u gornjem primjeru umjesto komprehencije liste koristili "map" funkciju, bilo bi potrebno napisati funkciju ili lambda izraz za kvadriranje:

```
>>> map(lambda x: x ** 2, range(5))
[0, 1, 4, 9, 16]
```

Korištenje komprehencije liste nije mnogo jednostavnije od korištenja "map" funkcije, ali ima bolje performanse. Primjerice, komprehencija liste izvodi se otprilike dvostruko brže od ekvivalentnih "for" petlji, a to je zbog toga što se komprehencija liste unutar interpretera izvodi brzinom programskog jezika C.

Komprehenciju liste možemo koristiti u kombinaciji sa "if" petljom, što je vrlo slično korištenju funkcije "filter":

```
>>> [x for x in range(10) if x % 2 == 0]
[0, 2, 4, 6, 8]
>>> filter(lambda x: x%2 == 0, range(10))
[0, 2, 4, 6, 8]
```

Napravimo sada izraz koji će vraćati kvadrate parnih brojeva između 1 i 10:

```
>>> [x ** 2 for x in range (1, 11) if x % 2 == 0]
[4, 16, 36, 64, 100]
```

Napravimo ekvivalentni izraz pomoću funkcije "map":

```
>>> map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, range(1, 11)))
[4, 16, 36, 64, 100]
```

Za implementaciju takvog izraza bilo je potrebno kombinirati funkcije "map" i "filter", pa je jasno vidljivo kako je u ovakvim slučajevima jednostavnije koristiti komprehenciju liste.

Iako komprehencija liste vraća listu, moguće je iteriranje preko različitih sekvenci:

```
>>> [x + y for x in 'abc' for y in ['01', '02', '03']]
['a01', 'a02', 'a03', 'b01', 'b02', 'b03', 'c01', 'c02', 'c03']
```

Korištenje izrazima orijentiranih funkcija u Pythonu skraćuje pisanje koda, povećava čitljivost, te ubrzava izvođenje programa, što je korisno kada je brzina izvođenja programa bitan faktor.

6. Primjeri korištenja lambda računa

U ovom poglavlju bit će nekoliko primjera u kojima će biti prikazano korištenje lambda izraza. Osim toga, bit će prikazano i korištenje funkcije "filter" u kombinaciji sa lambda izrazom, te test učinkovitosti takvog pristupa.

6.1. Učinkovitost funkcije "filter"

U ovom primjeru usporedit ćemo brzinu izvođenja funkcije "filter", koja je opisana u poglavlju 5.2, u odnosu na brzinu izvođenja funkcije koju ćemo sami definirati, a koja vraća isti rezultat.

Za potrebe testiranja napravit ćemo datoteku sa funkcijom "filter" koja vraća sve elemente liste koji su manji od 0, te datoteku u kojoj ćemo definirati vlastitu funkciju koja također vraća elemente liste manje od 0. Lista će sadržavati elemente između -10 i 10, a kako bi izvođenje trajalo duže, isti proces će se iterirati $2 \cdot 10^6$ puta. Datoteke ćemo pokretati naizmjenično, kako bi rezultati bili što mjerodavniji.

Možemo napraviti Python datoteku "trajanjeDef.py", koja sadrži "def" funkciju:

```
import time
pocetak = time.time()
lista = []

for y in range(0, 2000000):
  for x in range(-10, 11):
     if x < 0:
           lista.append(x)

kraj = time.time()
trajanje = kraj - pocetak
print "Program traje ", trajanje, " sekundi"
```

Trajanje izvođenja programa u gornjem primjeru mjerimo pomoću funkcije "time" iz modula "time".

Na sličan način možemo napraviti datoteku "trajanjeLambda.py":

```
import time
pocetak = time.time()

for y in range(0, 2000000):
  filter(lambda x: x <0, range(-10,11))
kraj = time.time()
trajanje = kraj - pocetak
print "Program traje ", trajanje, " sekundi"
```

Nakon 5 uzastopnih naizmjeničnih testiranja dobiveni su sljedeći rezultati:

Datoteka	1. testiranje	2. testiranje	3. testiranje	4. testiranje	5.testiranje
trajanjeDef.py	11.03	10.94	11.24	11.66	11.34
trajanjeLambda.py	9.27	9.36	9.25	9.45	9.14

Vidimo kako je korištenje funkcije filter uz lambda izraze brže nego korištenje funkcije koju smo sami definirali, što je korisno kod programa kod kojih je važna brzina izvođenja, ili kod programa koji primjenjuju takve funkcije nad velikim sekvencama.

Funkcije "map", "filter" i "reduce" ubrzavaju izvođenje programa, a naročito u kombinaciji sa lambda izrazima.

6.2. Lambda izrazi u Tkinter-u

[5] Tkinter je Pythonov de-facto standardni GUI[1] paket.

Možemo napraviti program koji će imati tri gumba: jedan sa crvenim tekstom, drugi sa plavim tekstom i treći sa zelenim tekstom. Klikom na određeni gumb, u konzoli će se ispisivati odgovarajuća poruka. U nastavku slijedi izvorni kod takvog programa:

```
import Tkinter as tkinter
class App:
  def __init__(self, parent):
     """Konstruktor"""
     frame = tkinter.Frame(parent)
     frame.pack()

     gumbCrveni = tkinter.Button(frame, fg = "red",
        text="crveni", command=self.gumbCmdCrveni)
     gumbCrveni.pack(side=tkinter.LEFT)

     gumbPlavi = tkinter.Button(frame, fg= "blue",
        text="plavi", command=self.gumbCmdPlavi)
     gumbPlavi.pack(side=tkinter.LEFT)

     gumbZeleni = tkinter.Button(frame, fg= "green",
        text="zeleni", command=self.gumbCmdZeleni)
     gumbZeleni.pack(side=tkinter.LEFT)

  def gumbCmdCrveni(self):
     self.printajBoju("crveni")

  def gumbCmdPlavi(self):
     self.printajBoju("plavi")

  def gumbCmdZeleni(self):
     self.printajBoju("zeleni")

  def printajBoju(self, bojaGumba):
     print "Pritisnuli ste %s gumb" % bojaGumba
```

[1]GUI - od eng. "Graphical User Interface", grafičko korisničko sučelje

```
if __name__ == "__main__":
    root = tkinter.Tk()
    app = App(root)
    root.mainloop()
```

Nakon što pokrenemo program, pojavljuje se sljedeći prozor:

Slika 6.1: Tkinter prozor

Klikom na određeni gumb, u konzoli se ispisuje odgovarajuća poruka:

```
Pritisnuli ste crveni gumb
Pritisnuli ste plavi gumb
Pritisnuli ste zeleni gumb
```

Gornji primjer implementirali smo korištenjem metoda koja se pozivaju prilikom akcije klika na gumb. Međutim, to je moguće implementirati i korištenjem lambda izraza umjesto metoda:

```
import Tkinter as tkinter
class App:
  def __init__(self, parent):
     """Konstruktor"""
     frame = tkinter.Frame(parent)
     frame.pack()

     gumbCrveni = tkinter.Button(frame, fg = "red",
        text="crveni", command=lambda: self.printajBoju("crveni"))
     gumbCrveni.pack(side=tkinter.LEFT)

     gumbPlavi = tkinter.Button(frame, fg= "blue",
        text="plavi", command=lambda: self.printajBoju("plavi"))
     gumbPlavi.pack(side=tkinter.LEFT)

     gumbZeleni = tkinter.Button(frame, fg= "green",
        text="zeleni", command=lambda: self.printajBoju("zeleni"))
     gumbZeleni.pack(side=tkinter.LEFT)
```

```
  def printajBoju(self, bojaGumba):
     print "Pritisnuli ste %s gumb" % bojaGumba

if __name__ == "__main__":
   root = tkinter.Tk()
   app = App(root)
   root.mainloop()
```

Umjesto definiranja metoda, koristili smo lambda izraze, te smo na taj način skratili pisanje koda, povećali njegovu čitljivost, te ubrzali izvođenje programa.

6.3. Primjer sa wxPython-om

[5] WxPython je besplatan program, koji je zapravo alternativa Tkinter-u. Nije uključen u Python, već ga je potrebno posebno instalirati. U prethodnom primjeru napravili smo jednostavan program koji ispisuje poruku ovisno o gumbu kojeg smo pritisnuli. Istu funkcionalnost možemo implementirati u wxPython-u, ali o ovom primjeru ćemo gumbima dati nazive "prvi" i "drugi". U nastavku je kod za wxPython:

```
import wx

class DemoFrame(wx.Frame):

   def __init__(self):
      """Konstruktor"""
      wx.Frame.__init__(self, None, wx.ID_ANY,
                  "wx lambda primjer",
                  size=(250,60)
                  )
      panel = wx.Panel(self)

      prviGumb = wx.Button(panel, label="prvi")
      prviGumb.Bind(wx.EVT_BUTTON, lambda dogadaj,
         naziv=prviGumb.GetLabel(): self.ispisi(dogadaj, naziv))
      drugiGumb = wx.Button(panel, label="drugi")
      drugiGumb.Bind(wx.EVT_BUTTON, lambda dogadaj,
         naziv=drugiGumb.GetLabel(): self.ispisi(dogadaj, naziv))
```

```
        sizer = wx.BoxSizer(wx.HORIZONTAL)
        sizer.Add(prviGumb, 0, wx.ALL, 5)
        sizer.Add(drugiGumb, 0, wx.ALL, 5)
        panel.SetSizer(sizer)

    def ispisi(self, dogadaj, labelaGumba):
        print "Pritisnuli ste %s gumb!" % labelaGumba

if __name__ == "__main__":
    app = wx.PySimpleApp()
    frame = DemoFrame().Show()
    app.MainLoop()
```

U ovom primjeru zanimljivo nam je vezanje događaja na gumb, zato jer u tom dijelu koristimo lambda izraz:

```
prviGumb.Bind(wx.EVT_BUTTON, lambda dogadaj, naziv=prviGumb.GetLabel():
    self.ispisi(dogadaj, naziv))
```

U navedenoj liniji koda na prvi gumb vežemo događaj, kojeg definiramo pomoću lambda izraza. Parametri lambda izraza su "dogadaj" i "naziv". Prvi parametar nam u ovom slučaju označava događaj, odnosno "klik" gumba, dok drugi parametar predstavlja labelu prvog gumba, koju dohvaćamo pomoću "GetLabel()" funkcije. Unutar lambda funkcije poziva se metoda "ispisi", koja ispisuje poruku, a kao parametre joj šaljemo "dogadaj" i "naziv".

Program funkcionira tako da se klikom na određeni gumb prikazuje odgovarajuća poruka:

```
Pritisnuli ste prvi gumb!
Pritisnuli ste drugi gumb!
```

U ovom primjeru mogli smo koristiti definicije funkcija za vezanje događaja klika gumba, ali smo to pojednostavnili korištenjem lambda izraza.

7. Zaključak

U prvom dijelu ovog rada ukratko je opisana povijest razvoja notacije za proizvoljne funkcije, te povijest funkcionalnog programiranja, a ponajviše lambda računa, čiji utemljitelj je Alonzo Church. Napravljen je i kratki uvod u funkcionalno programiranje. Drugo poglavlje fokusirano je na formalni (matematički) opis lambda računa. Opisani su osnovni elementi lambda notacije, te osnovne operacije unutar lambda računa.

Međutim, glavno težište u ovom radu stavljeno je na primjenu lambda računa, odnosno lambda izraza, u programskom jeziku Python, gdje se kroz primjere pokazuje zašto je dobro koristiti lambda izraze u određenim situacijama.

Korištenje lambda računa u Pythonu je ograničeno na jedan izraz, odnosno nije moguće definirati funkciju pomoću dva ili više izraza, kao u programskom jeziku Lisp, a razlog tome leži u činjenici da je Lisp u potpunosti funkcionalni programski jezik, dok Python objedinjuje više pristupa. Drugi razlog je to što su lambda izrazi relativno kasno uvedeni u Python.

Mogli bismo reći da lambda račun nije potreban za programiranje u Pythonu. Međutim, postoje situacije u kojima je korištenje lambda izraza izrazito prikladno, naročito kada želimo skratiti pisanje koda ili ubrzati izvođenje programa. Isto tako, korištenje lambda izraza je korisno prilikom definiranja anonimnih funkcija koje namjeravamo koristiti samo jednom.

Možemo zaključiti da, iako nije nužno, korištenje lambda izraza u Pythonu može biti vrlo korisno.

Literatura

[1] Functions lambda.
http://www.bogotobogo.com/python/python_functions_lambda.php.

[2] Lambda, filter, reduce and map.
http://www.python-course.eu/lambda.php.

[3] Barendregt, H. and Barendsen, E. Introduction to lambda calculus. 2000.
http://www.cse.chalmers.se/research/group/logic/TypesSS05/Extra/geuvers.pdf.

[4] Cardone, F. and Hindley, J. History of lambda-calculus and combinatory logic. 2006.
http://maths.swan.ac.uk/staff/jrh/papers/JRHHislamWeb.pdf.

[5] Driscoll, M.
http://www.blog.pythonlibrary.org/2010/07/19/the-python-lambda/.

[6] Ferg, S.
http://pythonconquerstheuniverse.wordpress.com/2011/08/29/lambda_tutorial/.

[7] Jung, A. A short introduction to the lambda calculus. 2004.
http://cs.nyu.edu/courses/spring12/CSCI-GA.2110-001/docs/lambda-calculus.pdf.

[8] Kurtz, G. Stupid lambda tricks.
http://p-nand-q.com/python/stupid_lambda_tricks.html.