SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Proširivanje Pythona programskim jezicima C/C++

Ivo Majić
Voditelj: *Doc. dr. sc. Domagoj Jakobović*

Zagreb, svibanj 2012.

SADRŽAJ

1. Uvod

Programski jezik Python spada u skupinu skriptnih jezika koji je danas sve više korišten za razne namijene. Python je svoju popularnost stekao lakom čitljivošću teksta programa te dinamičkim tipiziranjem (eng. duck typing), što značajno ubrzava razvoj programa. Također jedna od velikih prednosti Pythona je njegova mogućnost povezivanja s drugim programskim jezicima te je poznat kao jezik za lijepljenje (eng. glue language). Umjesto da je sva željena funkcionalnost ugrađena u jezgru jezika, Python je dizajniran da bude proširiv. Ova mogućnost se osim za pristupanje funkcionalnostima dostupnim samo u obliku C/C++ (te Fortran, iako to nije predmet ovog rada) biblioteka, može iskoristiti za ubrzanje kritičnih dijelova programa pisanih u Pythonu.

Nakon uvodnog poglavlja slijedi poglavlje u kojem se opisuje Python/C API tj. biblioteka C funkcija koje se koriste prilikom izgradnje proširenja. U trećem poglavlju je opisano korištenje raznih alata, koji olakšavaju pristup postojećim bibliotekama ili automatiziraju pisanje proširenja. U četvrtom poglavlju je dana kratka analiza prednosti pisanja proširenja u C/C++ u smislu ubrzanja konačnog programa.

2. Python/C API

Programski jezik Python osmislio je 1990. godine Guido van Rossum po uzoru na programski jezik ABC. Interpreter programskog jezika Python implementiran je u nekoliko različitih programskih jezika poput Jave (Jython) i C#-a (IronPython), no najčešće korištena (te ujedno i standardna) implementacija je izvedena u programskom jeziku C (CPython). Python je zamišljen kao jako proširivi programski jezik, te većinu svoje popularnosti duguje velikom broju dostupnih proširenja (eng. modula). Tema ovog rada je pisanje proširenja koristeći programske jezike C i C++ upravo za CPython implementaciju koristeći dostupni Python/C API. Veliki broj proširenja koji su dio Pythonove standardne biblioteke, kod kojih je brzina izvođenja kritična, koriste ovu mogućnost. Python/C API sadržan je unutar $Python.h$ datoteke zaglavlja, koja dolazi kao dio svake CPython instalacije (unutar $include$ direktorija).

2.1. Primjer korištenja

Kako bi podržao pisanje proširenja (eng. modula), Python/C API (eng. Application Programming Interface) definira određeni broj funkcija, makroa i varijabli koje omogućavaju dvosmjernu komunikaciju između proširenja i Python interpretera. Cjelokupni API se može učiniti dostupnim u željenom C programu uključivanjem **Python.h** zaglavlja.

```
#include <Python.h>
```

U ovom poglavlju će kroz kratak primjer biti objašnjeni najosnovniji dijelovi API-ja, te kako kratku funkciju napisanu u programskom jeziku C učiniti dostupnom u Pythonu [3]. Kao primjer je uzeta jednostavna funkcija **zbroji** koja računa zbroj dva broja, te će biti smještena unutar modula **racunop**.

```
int zbroji(int broj1, int broj2) {
        return broj1+broj2;
}
```

Kako bi funkcija mogla biti pozivana iz Pythona prvo treba napisati omotač funkciju (eng. wrapper function). Omotač funkcija će obavljati pretvorbu ulaznih argumenata u odgovarajuće C tipove podataka, prosljeđivanje tako pretvorenih argumenata u željenu funkciju te pretvorbu rezultata natrag u željeni Python tip podataka.

```
static PyObject* racunop_zbroji(PyObject *self, PyObject *args) {
        int broj1, broj2;
        if (!PyArg_ParseTuple(args, "ii", &broj1, &broj2))
                return NULL;
        int rezultat = zbroji(broj1, broj2);
        return Py_BuildValue("i", rezultat);
}
```

Omotač funkcija je ono što će Python pozvati prilikom korištenja funkcije zbroji(), primjerice kada se pozove naredba racunop.zbroji(3, 5). Funkcija vraća rezultat tipa PyObject, što je osnovni tip podataka iz kojeg su izvedene sve podatkovne strukture u Pythonu. Argumenti funkcije se pritom predaju kao članovi nepromjenjive liste (eng. tuple). Kako bi iz takvog objekta dohvatili argumente te obavili pretvorbu u C tipove podataka koristimo funkciju $PyArg\_ParseTuple$.

```
PyArg_ParseTuple(PyObject *args, const char *format, ...)
```

Funkcija $PyArg\_ParseTuple$ osim pokazivača na objekt koji sadrži argumente prima i format string koji omogućava automatsku pretvorbu u odgovarajući C tip podataka. U ovom primjeru format string je 'ii' jer funkcija očekuje dva cjelobrojna (eng. integer) podataka. Popis najčešće korištenih formata dostupan je u tablici 2.1.

Tablica 2.1: Format string znakovi

Znak	**Opis**
i	pretvara Python integer u C int
c	pretvara Python string duljine 1 u C char
f	pretvara Python float u C float
d	pretvara Python float u C double
s	pretvara Python string u niz C charova
O	vraća pokazivač na PyObject bez pretvorbe

Funkciji također treba predati varijable odgovarajućeg tipa (u istom poretku kao u format stringu), koje će pohraniti dobivene vrijednosti.

Format znak 'O' se najčešće koristi kada funkcija ne očekuje tip podataka koji se može direktno pretvoriti u neki od dostupnih C tipova. Primjeri takvih struktura podataka su $PyListObject$ (koji implementira Python listu) te $PyDictObject$ (koji implementira Python rječnik (eng. dictionary)). Svaka od tih struktura podataka ima definiran skup funkcija koje se mogu koristiti za pristup podacima unutar njih. Popis nekoliko dostupnih $PyListObject$ funkcija dan je u tablici 2.2.

Tablica 2.2: PyListObject funkcije

Funkcija	**Opis**
PyList_Check(PyObject *p)	vraća True ako je objekt tipa PyListObject
PyList_Size(PyListObject *list)	vraća duljinu liste
PyList_GetItem(PyListObject *list, int i)	vraća i-ti element liste

Prilikom vraćanja rezultata dostupna je funkcija $Py\_BuildValue$, koja dani C tip podataka pretvara u odgovarajući Python tip podataka (također koristeći odgovarajući format string).

Kako bi Python interpreter znao koje su sve funkcije dostupne unutar određenog modula, sve moraju biti popisane u tablici metoda (pri čemu je svaka definicija tipa $PyMethodDef$).

```
static PyMethodDef racunop_methods[] = {
        {"zbroji", racunop_zbroji, METH_VARARGS, racunop_zbroji_doc},
        {NULL, NULL, 0, NULL} /* Sentinel */
};
```

U tablici metoda za svaku funkciju koja će biti dostupna u modulu treba navesti ime koje će se koristiti prilikom pozivanja u Pythonu (u ovom primjeru je to **zbroji**), stvarno ime omotač funkcije koja će biti pozivana (u ovom primjeru funkcija **racunop_zbroji**), koju vrstu argumenata funkcija prima (pomoću posebnih konstanti definiranih unutar biblioteke, opisanih u tablici 2.3) te kratak opis funkcije.

Tablica 2.3: Vrste argumenata

Konstanta	**Opis**
METH_VARARGS	ako funkcija pri pozivu prima argumente
METH_NOARGS	ako funkcija pri pozivu ne prima argumente

Na kraju treba dodati inicijalizacijsku funkciju koja će biti pozivana kada se modul importa u Python skripti. Ime inicijalizacijske funkcije mora biti formata **initIMEMODULA()** pri čemu ime mora odgovarati odabranom imenu proširenja (eng. modula). Unutar nje se poziva funkcija $Py\_InitModule$ koja kao argumente prima željeno ime proširenja, zatim tablicu metoda definiranu ranije te kratak opis proširenja.

```
PyMODINIT_FUNC initracunop(void) {
        Py_InitModule("racunop", racunop_methods, racunop_doc);
}
```

2.2. Prevođenje

Kako bi maksimalno olakšao prevođenje izvornog teksta proširenja u izvršni, Pythonov $distutils$ modul sadrži nekoliko pomoćnih klasa i funkcija. U nastavku je dana kratka skripta koja će obaviti pozivanje prevodioca sa svim potrebnim parametrima.

```
from distutils.core import setup
from distutils.core import Extension

module = Extension("racunop", sources = ["racunop.c"])
setup(name="racunop", version="1.0", ext_modules=[module])
```

Slika 2.1: setup.py

Nakon pokretanja skripte sa $python\ setup.py\ build$ i uspješnog prevođenja, stvoriti će se direktorij build koji će sadržavati gotovi izvršnp proširenje. Gotovo proširenje će ovisno o operacijskom sustavu biti u obliku *.so (na operacijskom sustavu Linux) ili *.pyd (Python DLL, na operacijskom sustavu Windows) izvršne biblioteke. Razlika između običnih izvršnih biblioteka te ovih biblioteka je postojanje funkcije **initIMEMODULA()**, koju će Python interpreter prvu pozvati prilikom uključivanja proširenja.

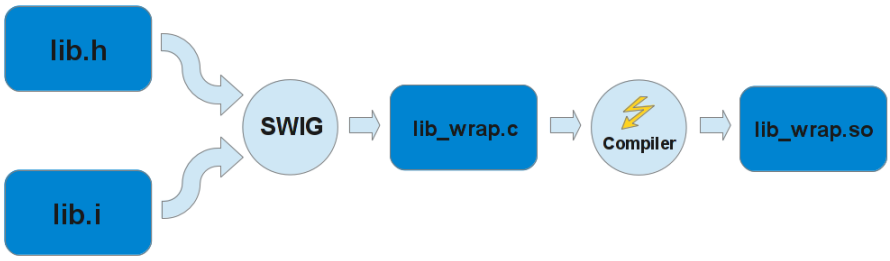
Proširenje se nakon toga može naredbom $import$ uključiti unutar željene Python skripte (u ovom primjeru $import\ racunop$).

3. Dodatni alati

Iz prethodnog poglavlja vidljivo je da ručno pisanje proširenja, koje bi sadržavalo velik broj funkcija (poput čitavih biblioteka) bilo jako dugotrajno. U svrhu lakšeg razvoja proširenja postoji nekoliko dodatnih alata koji na različite načine ubrzavaju taj postupak, neki od njih opisani su u ovom poglavlju.

3.1. SWIG

SWIG je razvojni alat čija je namjena olakšati stvaranje poveznica za razne programske jezike (ne samo skriptne poput Pythona) prema C i C++ bibliotekama [4]. SWIG je prevodioc koji iz C/C++ deklaracija funkcija (i klasa) generira odgovarajuće omotače, kako bi omogućio pristup tim funkcijama iz drugih programskih jezika. Među trenutno dostupne ciljne jezike spadaju Perl, Python, TCL, Ruby te drugi, no u ovom radu naglasak će biti stavljen na generiranje poveznica za programski jezik Python. Na slici 3.1 je prikazan općeniti postupak generiranja poveznica.



Slika 3.1: SWIG tok izvođenja

Osim datoteka zaglavlja (*.h, eng. header) koja sadrži deklaracije, SWIG-u je potrebno predati i posebnu konfiguracijsku datoteku (*.i, eng. interface) u kojoj je definirano ime proširenja te koji sve elementi iz datoteke zaglavlja moraju biti dostupni za korištenje u konačnom modulu. SWIG će generirani omotač (eng. wrapper) za željeni

ciljni jezik smjestiti u datoteku **IMEMODULA_wrap.cxx**, gdje se **IMEMODULA** uzima iz konfiguracijske datoteke. U primjeru u nastavku je prikazan cjelokupni postupak generiranja poveznica.

3.1.1. Primjer korištenja

U ovom primjeru bit će prikazan postupak izgradnje Python modula korištenjem SWIG-a. Kao u prethodnom poglavlju razvit ćemo modul **racunop** koji će u ovoj implementaciji sadržavati klasu **Brojevi** koja prilikom instanciranja prima dva cijela broja. Klasa pruža javne metode **zbroj()** (koja ispisuje zbroj ta dva broja) te **umnozak()** (koja ispisuje njihov umnožak). Izvorni programski tekst implementacije smješten je u datoteku **Brojevi.cpp**.

```
#include <stdio.h>
#include "Brojevi.h"

Brojevi::Brojevi(int broj1, int broj2) {
        this->broj1 = broj1;
        this->broj2 = broj2;
}

void Brojevi::zbroj() {
        printf("Zbroj je: %d\n", this->broj1+this->broj2);
}

void Brojevi::umnozak() {
        printf("Umnozak je: %d\n", this->broj1*this->broj2);
}
```

Slika 3.2: Brojevi.cpp

U datoteci zaglavlja **Brojevi.h** su smještene deklaracije konstruktora te dostupnih metoda. SWIG će prilikom izgradnje modula analizirati datoteku zaglavlja te na temelju deklaracija **javnih** metoda generirati omotače za njih koristeći Python/C API opisan ranije. Dodatne konfiguracijske postavke (poput imena modula) se smještaju u datoteku **racunop.i**. U konfiguracijskoj datoteci se korištenjem ključne riječi **%module** definira željeno ime modula. Unutar **%{...}** odsječka se zatim dodaju dodatne datoteke zaglavlja i programski tekst koji je potreban za **prevođenje** generirane datoteke s omotačima.

```
class Brojevi {
        int broj1;
        int broj2;
public:
        Brojevi(int broj1, int broj2);
        void zbroj();
        void umnozak();
};
```

Slika 3.3: Brojevi.h

```
%module racunop

%{
#include "Brojevi.h"
%}

%include "Brojevi.h"
```

Slika 3.4: racunop.i

U sljedećem odsječku pomoću ključne riječi **%include** dodajemo datoteke zaglavlja koje će SWIG analizirati te generirati omotače za funkcije i klase sadržane unutar njih.

Sada se poziva SWIG naredbom (u ovom primjeru naredba se poziva sa zastavicom $-python$ kako bi se generirale Python poveznice):

```
swig -python -c++ racunop.i
```

Ova naredba će generirati omotače te ih smjestiti u datoteku **racunop_wrap.cxx**. Zatim koristeći odgovarajući prevodioc prevodimo racunopp.cpp te racunop_wrap.cxx u izvršni program. Prilikom prevođenja **racunop_wrap.cxx** treba uključiti i odgovarajući direktorij koji sadrži Python.h datoteku zaglavlja (Python/C API).

```
g++ -c racunop.cpp
g++ -I/usr/include/python2.7 -c racunop_wrap.cxx
```

Na kraju sve rezultirajuće izvršne datoteke treba povezati u jednu *.so datoteku koja predstavlja sam modul. Pri tome treba uključiti i sve dodatne izvršne biblioteke koje smo koristili (u ovom slučaju libstdc++ koji sadrži funkciju printf()).

```
ld -shared -o _racunop.so racunop.o racunop_wrap.o libstdc++.so
```

Ovako izgrađeni modul sada se može pozvati unutar Pythona te instancirati klasu Brojevi.

```
>> import racunop

>> var = racunop.Brojevi(3,5)
>> var.zbroj()
Zbroj je: 8
>> var.umnozak()
Umnozak je: 15
```

3.2. Ctypes

Ctypes je programski modul koji je dio Pythonove standardne biblioteke, koji omogućava direktno pozivanje funkcija iz izvršnih biblioteka [2]. Ova mogućnost čini ctypes idealnim alatom za pisanje poveznica prema vanjskim C bibliotekama koristeći samo Python. Trenutni nedostatak je nepostojanje podrške za C++ klase, zbog različitih implementacija C++ standarda koje koriste pojedini prevodioci.

Slika 3.5: Ctypes tok izvođenja

Podržano je nekoliko tipova izvršnih biblioteka (kako je prikazano u tablici 3.1), te je svaka implementiran pomoću istoimene klase. Svaka od njih ima definiranu statičku metodu $LoadLibrary(name)$ gdje $name$ predstavlja ime biblioteke koju želimo koristiti.

Tablica 3.1: Podržane izvršne biblioteke

DLLTYPE	**Opis**
CDLL	funkcije unutar ove vrste biblioteka koriste C standard pozivanja te vraćaju vrijednost tipa int
OleDLL	funkcije unutar ove vrste biblioteka koriste $stdcall$ standard pozivanja te vraćaju $HRRESULT$ kod (specifično za operacijski sustav Windows)
WinDLL	funkcije unutar ove vrste biblioteka koriste $stdcall$ standard pozivanja te vraćaju vrijednost tipa int

Iz ovako učitane biblioteke sada se može pozivati bilo koja u njoj sadržana funkcija. Prilikom predaje argumenata funkcijama, ctypes automatski vrši pretvorbu u odgovarajuće C tipova podataka (isto vrijedi i za povratne vrijednosti).

3.2.1. Primjer korištenja

Kao primjer korištenja ctypes modula u nastavku je dan kratki programski odsječak koji prikazuje implementaciju $sleep()$ funkcije. Sama implementacija funkcije nalazi se unutar $libc$ biblioteke (na operacijskom sustavu Linux).

```
import ctypes

biblioteka = ctypes.cdll.LoadLibrary("libc.so")

def sleep(sekundi):
    print "Spavam %d sekundi" % sekundi
    biblioteka.sleep(sekundi)

'''Testiranje'''
sleep(5)
```

Slika 3.6: Ctypes primjer

Biblioteka $libc$ je tipa CDLL pa ju učitavamo koristeći $LoadLibrary()$ metodu istoimene klase koja će vratiti objekt tipa $LibraryLoader$. Kako se $libc$ biblioteka nalazi unutar standardnog direktorija za pohranu izvršnih biblioteka, potrebno je navesti samo ime biblioteke. U slučaju da se biblioteka nalazila u nekom drugom direktoriju, bilo bi potrebno napisati punu putanju.

Funkcije sadržane unutar biblioteke sada se mogu pozivati poput običnih Python metoda. U ovom primjeru se tako $sleep()$ funkcija preslikava u metodu $sleep()$ unutar objekta $biblioteka$ koji je tipa $LibraryLoader$.

3.3. Cython

Cython je prevodioc (eng. compiler) koji proširuje sintaksu programskog jezika Python s mogućnošću statičkog definiranja C tipova podataka [1]. Tako napisan tekst programa prilikom prevođenja može iskoristiti sve optimizacije koje koriste obični C/C++ prevodioci. Ovo omogućuje da se sav tekst programa piše koristeći Python sintaksu i pripadne tipove podataka, a kritične dijelove koristeći C tipove podataka koje nudi Cython. Proširenja prevedena u izvršni program korištenjem Cythona se zatim mogu pomoću $import$ naredbe koristiti u Pythonu.

Slika 3.7: Cython tok izvođenja

Osim ubrzavanja postojećih programa pisanih Pythonu, Cython omogućuje direktno pozivanje C/C++ funkcija i klasa. Neke od ovih mogućnosti su opisane kroz primjer dan u nastavku.

3.3.1. Primjer korištenja

Kao primjer uzimamo funkciju $prosti(N)$ koja vraća prvih N (ograničeno na $N = 30000$) prostih brojeva u obliku Python liste (cijeli tekst programa prikazan je pod 3.8).

U linijama 2-3 varijable deklariramo statički koristeći Cythonovu $cdef$ naredbu, koja omogućava korištenje C tipova podataka. Zbog ove promjene će sve petlje koje u sebi ne sadrže referencu na neki Python objekt, prilikom prevođenja biti pretvorene u C petlje (linije 11-12). Linije 11-12 u ovom primjeru predstavljaju kritični dio programa, koji uzima najviše vremena prilikom izračuna.

```
1  def prosti(int N):
2      cdef int n, k, i
3      cdef int p[30000]
4      rezultat = []
5      if N > 30000:
6          N = 30000
7      k = 0
8      n = 2
9      while k < N:
10         i = 0
11         while i < k and n % p[i] != 0:
12             i = i + 1
13         if i == k:
14             p[k] = n
15             k = k + 1
16             rezultat.append(n)
17         n = n + 1
18     return rezultat
```

Slika 3.8: prosti.pyx

Prevođenje ovako napisanog teksta programa obavlja se koristeći pomoćne funkcije iz Cythonovog *distutils* modula (slično kao prilikom prevođenja ručno pisanih proširenja korištenjem Python/C API-ja). U nastavku je dan sav programski tekst potreban za prevođenje, skripta se pokreće naredbom *python setup.py build_ext*.

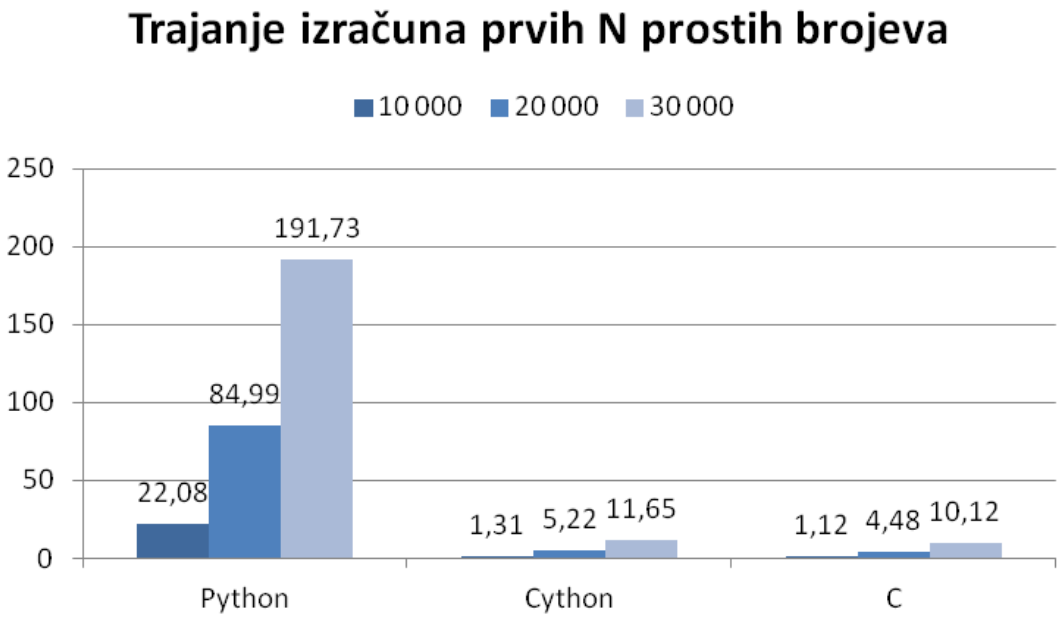
```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("prosti", ["prosti.pyx"])]
)
```

Slika 3.9: setup.py

4. Ubrzanje

Jedan od razloga pisanja proširenja koristeći programske jezike C/C++ jest ubrzavanja određenih segmenata programa pisanih u Pythonu. Kao primjer uzimamo algoritam za izračun prvih N prostih brojeva opisan ranije (izvorni programski tekst 3.8). Za N su izabrane vrijednosti od [10 000, 20 000, 30 000], te je algoritam implementiran u Pythonu, Cythonu i C-u.



Slika 4.1: Trajanje izračuna prvih N prostih brojeva (u sekundama)

Iz grafa 4.1 je vidljivo da je Python implementacija najsporija, dok najviše iznenađuju rezultati Cython implementacije u odnosu na C implementaciju. U Cython implementaciji je ovakvo ubrzanje postignuto sa samo dvjema $cdef$ naredbama, zbog kojih su kritične petlje pretvorene u C petlje prilikom prevođenja.

5. Zaključak

Programski jezik Python je danas jedan od najraširenijih skriptnih jezika. Njegov dostupni C API omogućava razvoj proširenja koristeći programske jezike C i C++, te time omogućuje korištenje Pythona u velikom broju situacija.

Osim samog API-ja razvijeno je mnogo alata koji svaki na svoj način olakšavaju suradnju između Pythona i C-a. SWIG alat primjerice predstavlja opciju kada imamo gotov C/C++ programski tekst koji želimo koristiti direktno iz Pythona. Ctypes modul je koristan za brz pristup C funkcijama unutar gotovih izvršnih biblioteka (primjerice upravljačkih funkcija za vanjske uređaje, koje su često pisane u C-u). Cython pak predstavlja zlatnu sredinu jer pruža mogućnost pisanja proširenja korištenjem Python sintakse, uz mogućnost ubrzavanja kritičnih dijelova korištenjem C tipova podataka.

Cilj ovog rada je bio dati pregled ova tri različita pristupa integracije C i C++ programskog teksta s Pythonom. Na kraju je na korisniku da odabere koji od ovih pristupa najbolje rješava zadani problem.

6. Literatura

[1] Cython Community. Cython documentation. http://docs.cython.org, 2012.

[2] Python Community. Ctypes — a foreign function library (module) for Python. http://docs.python.org/library/ctypes.html, 2012.

[3] Python Community. Python documentation. http://docs.python.org, 2012.

[4] SWIG Community. Swig documentation. http://www.swig.org/Doc2.0, 2011.

7. Sažetak

U ovom radu je opisan postupak izgradnje proširenja (modula) za programski jezik Python korištenjem programskih jezika C/C++. Opisan je postupak ručnog pisanja proširenja (eng. module) korištenjem Python/C API-ja. Prikazan je i postupak automatiziranog generiranja poveznica (eng. bindings) korištenjem SWIG alata. Za direktan pristup funkcijama unutar izvršnih biblioteka opisan je Ctypes modul, koji je dio Pythonove standardne biblioteke. Treći opisani alat je Cython koji u Python dodaje mogućnost statičkog deklariranja varijabli. Prikazana je i kratka analiza prednosti pisanja proširenja koristeći opisane alate, u smislu ubrzanja konačnog programa.