Kurs rozszerzony języka Python

Inne języki

Marcin Młotkowski

4 stycznia 2022

Plan wykładu

Plan wykładu

Kanoniczna implementacja

CPython

Podstawowa implementacja języka Python w C.

PyPy

- jit compilation;
- napisany w RPython (Restricted Python);
- wysoka zgodność z Pythonem 2.7 i 3.8;
- możliwość dołączania własnego odśmiecacza pamięci;
- wsparcie dla *greenletów* i stackless;
- nieco inne zarządzanie pamięcią.

Stackless Python

- interpreter oparty na mikrowątkach realizowanych przez interpreter, nie przez kernel;
- dostępny w CPythonie jako *greenlet*;
- *stackless* bo unika korzystania ze stosu wywołań C.

Cython

Inspirowany składnią C język podobny do pythona (nadzbiór Pythona).
Kod jest kompilowany do C/C++ i dostępny dla CPythona jako moduł.
Realizacje: pandas

Numba

Podzbiór Pythona kompilowany LLVM.

Jython

Cechy Jythona

- implementacja Pythona na maszynę wirtualną Javy;
- kompilacja do plików `.class`;
- dostęp do bibliotek Javy;
- zgodny z Python 2.7.1.

IronPython

- Implementacja Pythona w środowisku Mono i .NET;
- zgodny z Pythonem 2.7 i 3.4 (alpha1).

Python for S60

Implementacja Nokii na tefony komórkowe z systemem Symbian 60

- implementacja Python wersji 2.2.2;
- dostęp do sprzętu (SMS'y, siła sygnału, nagrywanie video, wykonywanie i odbieranie połączeń);
- wsparcie dla GPRS i Bluetooth;
- dostęp do 2D API i OpenGL.

Ostateczne rozwiązanie

Zaprogramować w C i udostępnić w Pythonie jako moduł.

Ostateczne rozwiązanie

Zaprogramować w C i udostępnić w Pythonie jako moduł.
Co jest potrzebne: C API

Problemy łączenia dwóch języków

Zagadnienia

- problemy z różnymi typami danych (listy, kolekcje, napisy);
- przekazywanie argumentów i zwracanie wartości;
- tworzenie nowych wartości;
- obsługa wyjątków;
- zarządzanie pamięcią.

Dodanie do Pythona nowej funkcji

Zadanie

Implementacja obliczania n-tej liczby Fibonacciego w C

Dodanie do Pythona nowej funkcji

Zadanie

Implementacja obliczania n-tej liczby Fibonacciego w C

Elementy implementacji:

- plik nagłówkowy <Python.h>;
- implementacja funkcji;
- odwzorowanie funkcji w C na nazwę udostępnioną w Pythonie;
- funkcja inicjalizująca o nazwie *initnazwa_modułu*.

Implementacja funkcji (1)

```c
#include <python3.8/Python.h>
extern PyObject * fib(PyObject *, PyObject *);

PyObject * fib(PyObject * self, PyObject * args)
{

  PyObject * res;
  PyObject * wyraz;

 if (!PyArg_ParseTuple(args, "O", &wyraz))
  {
      printf("Niepoprawne argumenty\n");
      return NULL;
  }
```

Implementacja funkcji (2)

```
n = PyLong_AsLong(wyraz);

if (n == 0)
{
  res = Py_BuildValue("i", 0);
  Py_INCREF(res);
  return res;
}
```

Implementacja funkcji (3)

```
    ...
    res = Py_BuildValue("i", w11);
    Py_INCREF(res);
    return res;
}
```

Deklaracje modułu

```
static PyMethodDef metody[] = {
  {"cfib", fib, METH_VARARGS,
       "n-ta liczba Fibonacciego", },
  { NULL, NULL, -1, NULL }
};

static PyModuleDef moduledef = {
  PyModuleDef_HEAD_INIT,
  "fastcomp",
  "Szybkie obliczenia",
  -1,
  metody,
  NULL, NULL, NULL, NULL,
};
```

Inicjowanie modułu

```
PyMODINIT_FUNC
PyInit_fastfibb(void)
{
  PyObject *m;
  m = PyModule_Create(&moduledef);

  return m;
}
```

Kompilacja i instalacja

setup.py

```
from distutils.core import setup, Extension

modul = Extension('fastfibb',
                  sources = ['fastfb.c'])

setup (name = 'fastfibb',
       version = '0.1',
       description = 'This is a demo package',
       ext_modules = [modul])
```

Kompilacja i instalacja

```
$ python setup.py build
$ python setup.py install
```

Typy danych w Pythonie

Wszystko w Pythonie jest obiektem

Zarządzanie pamięcią

Mechanizm zarządzania pamięcią

- Każdy obiekt ma licznik odwołań zwiększany za każdym przypisaniem.
- Jeśli licznik jest równy zero obiekt jest usuwany z pamięci.
- W programach w C trzeba dbać o aktualizację licznika.

Zmiana licznika odwołań

Zwiększenie licznika

void Py_INCREF(PyObject *o)

Zmniejszenie licznika

void Py_DECREF(PyObject *o)

Trochę łatwiej

Biblioteka Boost:

+ łączenie Pythona z C++
+ łatwiejsza od C API
  - czasem nie da się ominąć C API (ale się rozwija)

Jak skorzystać

Skopiować do lokalnego katalogu plik *.so

```
from fastfibb import cfib
```

Wykonanie programów Pythonowych

```
Py_Initialize();
PyRun_SimpleString("i = 2")
PyRun_SimpleString("i = i*i\nprint(i)")
Py_Finalize();
```

Wykonanie programów w pliku

```
Py_Initialize();
FILE * f = fopen("test.py", "r");
PyRun_SimpleFile(f, "test.py");
Py_Finalize();
```

Kompilacja

```
gcc -lpython3.8 test.c
```

Bezpośrednie wywoływanie funkcji Pythonowych

Deklaracja zmiennych

```
PyObject *pName, *pModule, *pArgs, *pFunc, *pValue;
```

Import modułu Pythonowego

```
Py_Initialize();
pName = PyString_FromString("modulik");
pModule = PyImport_Import(pName);
```

Pobranie funkcji z modułu

```
pFunc = PyObject_GetAttrString(pModule, "foo");
```

Wywołanie funkcji

```
pValue = PyObject_CallObject(pFunc, pArgs);
```

Plan wykładu

Lokalne środowisko Pythonowe

virtualenv

Tworzy w lokalnym katalogu pełną wersję środowiska pythonowego, którą można modyfikować niezależnie od głównej instalacji. Można mieć wiele takich wirtualnych środowisk.

Lokalne środowisko Pythonowe

virtualenv

Tworzy w lokalnym katalogu pełną wersję środowiska pythonowego, którą można modyfikować niezależnie od głównej instalacji. Można mieć wiele takich wirtualnych środowisk.

```
$ virtualenv --system-site-packages $HOME/mojesrodowisko
$ cd $HOME/mojesrodowisko/
$ source bin/activate
```

Przykład

jupyter

Interaktywne środowisko do analizy danych i obliczeń naukowych, np. w pythonie.

Przykład Pawła Rychlikowskiego

Plan wykładu

Formaty

- egg: stary format;
- wheel: aktualny.

Formaty

- egg: stary format;
- wheel: aktualny.

Instalacja pakietów

pip

Dystrybucja programów

- Cyton: wygenerowanie kodu w C i kompilacja;
- Nuitka: generowanie kodu C++;

Dystrybucja programów

- Cyton: wygenerowanie kodu w C i kompilacja;
- Nuitka: generowanie kodu C++;
- inne, np. py2exe

A bez kompilacji

Skompresować pliki do zip'a!

A bez kompilacji

Skompresować pliki do zip'a!

1. sposób

Plik początkowy nazwać `__main__.py` i skompresować cały projekt.

A bez kompilacji

Skompresować pliki do zip'a!

1. sposób

Plik początkowy nazwać `__main__.py` i skompresować cały projekt.

2. sposób

```
$ python3 -m zipapp apka -m ''apka:startapp''
```

gdzie `apka` to katalog z plikami, a plik `apka/startapp.py` to początek programu.

Plan wykładu

Rozszerzanie właściwości funkcji

```
def szalenie_skomplikowana_funkcja(arg1, arg2, arg3):
    ...
```

Śledzenie wywołania funkcji

Chcemy śledzić wywołania zaimplementowanych funkcji, tj. informacje o wywołaniu oraz informacja o argumentach wywołania. Bez ingerowania w te funkcje.

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print "Wywoływana funkcja: foo z argumentami", args
    return foo(*args)
```

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print "Wywoływana funkcja: foo z argumentami", args
    return foo(*args)
```

Co z tym zrobić 1.

Zamiast *foo* używamy *log_foo*.

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print "Wywoływana funkcja: foo z argumentami", args
    return foo(*args)
```

Co z tym zrobić 1.

Zamiast *foo* używamy *log_foo*.

Co z tym zrobić 2.

foo = log_foo

Uniwersalna funkcja opakowująca inne funkcje

```
def log(fun):
    def opakowanie(*args):
        print "funkcja:", fun.__name__, "argumenty", args
        return fun(*args)
    return opakowanie
```

Uniwersalna funkcja opakowująca inne funkcje

```
def log(fun):
    def opakowanie(*args):
        print "funkcja:", fun.__name__, "argumenty", args
        return fun(*args)
    return opakowanie
```

Zastosowanie

foo = log(foo)

Dekoratory

```
def log(fun):
    def opakowanie(*args):
        print(f"{fun.__name__} {args}")
        return fun(*args)
    return opakowanie
```

Dekoratory

```
def log(fun):
    def opakowanie(*args):
        print(f"{fun.__name__} {args}")
        return fun(*args)
    return opakowanie
```

Zastosowanie

```
@log
def foo(args):
    ...
```

Pomiar czasu wykonania funkcji

```
def wydajnosc(fun):
    def opakowanie(*args):
        czas = timeit.timeit(lambda : fun(*args), number=1(
        wynik = fun(*args)
        print(f"{fun.__name__} {args} {czas}")
        return wynik
    return opakowanie
```

Przykład z życia: cache pytań SQL

```
def cache_query(func):
    """Keszowanie zapytań"""
    def wrapper(conn, query, typ):
        # inicjowanie cache
        db = shelve.open("/tmp/query.cache")
        if query in db:
            res = db[query]
            return res
        res = func(conn, query, typ)
        db[query] = list(res)
        db.sync()
        res = db[query]
        return res
    return wrapper
```

Dekoratory standardowe

Dekorowanie programów wielowątkowych

```
from threading import Lock
my_lock = Lock()

@synchronized(my_lock)
def critical1(): ...

@synchronized(my_lock)
def critical2(): ...
```

Implementacja dekoratora

```
def synchronized(lock):
    def wrap(f):
        def new_function(*args, **kw):
            lock.acquire()
            try:
                return f(*args, **kw)
            finally:
                lock.release()
        return new_function
    return wrap
```