19. C + Python

14 декември 2023

Преговор - модули

Какво може да бъде модул в python?

- Файл с разширение `.py`
- Директория съдържаща файл с име `__init__.py`

Преговор - байткод

Как може да видим кода на функция в Python?

- През `__code__` атрибута:

```
>>> (lambda: 42).__code__
<code object <lambda> at 0x7fc717a0dea0, file "<stdin>", line 1>
```

Builtin

Ho...

```
>>> print.__code__
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'builtin_function_or_method' object has no attribute
'__code__'
>>> print
<built-in function print>
>>> print.__class__
<class 'builtin_function_or_method'>
```

Защо?

- Повечето Пайтън е написан на Пайтън
- ... например `datetime`, `functools`, etc.
- Но и доста е написано на C
- ... например `print`
- Понякога се прави за скорост
- Понякога се прави, защото няма друг начин

код?

код!

Специални функции в C кода

Забелязахте ли pattern-а в имената на функциите в C кода?

- `PyObject_print`, `PyObject_Str`, `PyErr_format`, …
- pattern-а е `Py(Type)_(Function)`

Не се отнася само за функции

... Цели типове са имплементирани на C

```
>>> int
<class 'int'>
>>> int.x = 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't set attributes of built-in/extension type 'int'
```

Още код!

Вградени типове

- `int, float, str, list, dict, set`
- `map` и `filter` също са вградени типове, а не фунцкии
- има още неща в `__builtins__`

Още примери за C функции

- `PyDict_New`, `PyDict_Contains`, `PyDict_SetItem`
- `PyDict_SetItemString`, `PyDict_GetItem`
- `PyDict_GetItemString`, `PyDict_Size`, `PyDict_Merge`, `PyDict_Update`
- `PyList_New`, `PyList_Size`, `PyList_GetItem`
- `PyList_SetItem`, `PyList_Insert`, `PyList_Append`
- `PyList_GetSlice`, `PyList_Sort`, `PyList_Reverse`
- и т.н.

С кодът зад Python

- Чрез тези функции е имплементиран Пайтън
- Това е т.нар. Python C API
- ... ние можем също да ги използваме
- built-in (вграден) е подвеждащо име
- ... ние също можем да си дефинираме built-in функции

Какво означава “вграден”

- Всичко вградено ли се намира в `__builtins__`?

```
import _sqlite3

_sqlite3.connect
# <built-in function connect>
```

- Малко повече за този модул...

```
_sqlite3.__file__
# '/usr/lib/python3.11/lib-dynload/_sqlite3.cpython-311-x86_64-linux-gnu.so'
# ... или под Windows:
# 'C:/python3.11/DLLs/_sqlite3.pyd'
```

Модули - втори път

Всъщност модул в python може да бъде:

- Файл с разширение `.py`
- Директория съдържаща файл с име `__init__.py`
- Файл с разширение `.pyd/.so`

Native модули

- `.pyd` е преименуван `.dll` файл (dynamic link library)
- `.so` си е `.so` (shared object)
- Два термина за едно и също
- Представляват половината от C библиотека
- Другата половина е `.h` (header) файла

Какво съдържа един .dll/.so

- Съдържа изпълним код, import таблица, export таблица
- Последната съдържа имената на функциите
- Не се съдържат, обаче, дефиниции на типове
- Не се съдържат аргументите на функциите
- Как тогава можем да компилираме код, който използва функции от такива DLL-и?
- С header файлове. Те съдържат дефинициите на типове, декларации на функции и т.н.

Демо - C стандартна библиотека

- Под Windows: `dumpbin /exports C:\Windows\System32\msvcrt.dll`
- Под Linux: `nm -gD nm -gD /usr/lib/libc.so.6`

Демо - Python C API

- Под Windows: `dumpbin /exports C:\Python3.11\DLLs\python3.dll`
- Под Linux: `nm -gD nm -gD /usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0`

Python C API - хедъри

В общи случай се намират:

- Под Linux: `/usr/include/python{version}/`
- Под Windows: `C:\python{version}\include`

Python.h

- Декларации на функции
- Разни пре-процесори, които трябват на Python

C vs Python

- `int` в C не е като `int` в Пайтън
- `string` в C не е като `string` в Пайтън
- Когато искаме да подаваме стойности от едното място към другото, се налага да правим някакво преобръщане
- ... то се нарича **marshalling**

Marshalling към Python

```
Py_BuildValue("s", "spam") -> 'spam'
Py_BuildValue("i", 42) -> 42
Py_BuildValue("(sii)", 42, "hi", 8) -> (42, 'hi', 8)
Py_BuildValue("{is,is}", 1, "one", 2, "two") -> {1: 'one', 2: 'two'}
Py_BuildValue("") -> None
```

Marshalling към C

```
const char* str;
int number;
PyArg_ParseTuple(args, "si:string_peek", &str, &number);
```

Защо C API?

- Пайтън е бавен! (Numpy, Pandas)
- Преизползване на примитиви на OS (posix, win32, Cocoa + pyobjc)
- Преизползване на готов код (MySQL, PostgrSQL, Qt, PyGTK+,)
- Достъп до хардуер (Tensorflow, PyTorch, PyOpenGL, vulkan)
- Искам си указателите!

Cython - алтернатива на C API

- Позволява ни да пишем [почти] Python
- Сорс файлове с разширение `.pyx`
- Произвежда native (`.pyd`/`.so`) модул
- Вика Python C API без да пишем C код

Cython пример

```
# fib.pyx

def fib(n):
    """Print the Fibonacci series up to n."""
    a, b = 0, 1
    while b < n:
        print(b)
        a, b = b, a + b
```

Cython компилация като пакет

- `$ pip install cython`
- обикновено го слагаме в пакет (който качваме pip)
- Във файл с име `setup.py`:

```
from setuptools import setup
from Cython.Build import cythonize

setup(
    ext_modules=cythonize("fib.pyx"),
)
```

- `$ python setup.py build_ext --inplace`

Cython - компилация в работна среда

- Има алтернативен вариант (удобен при честа промяна на кода)
- В същата директория като `fib.pyx` правим `fib.pyxbld`:

```
from distutils.extension import Extension

def make_ext(modname, pyxfilename):
    return Extension(name=modname, sources=[pyxfilename], language='c')
```

- В python:

```
>>> import pyximport
>>> pyximport.install(build_dir = '/some/path')
>>> import fib
>>> fib
<module 'fib' from '[...]/fib.cpython-38-x86_64-linux-gnu.so'>
```

Cython - компилация в работна среда (2)

- Работи тривиално в *nix
- Може да се окаже по-пипкаво за подкарване в Window
- Работи, защото import системата в python е pluggable
- Можем да си регистрираме свои разширения за import
  (в случая `.pyxbld`)
- За сравнение – Hy – Lisp диалект работещ върху Python
  https://docs.hylang.org/en/stable/interop.html#using-hy-from-python

Cython - компилация в работна среда (3)

```
>>> fib.fib
<cyfunction fib at 0x7f368b750380>
>>> fib.fib(7)
0
1
1
2
3
5
```

Cython – output

- `.pyd/.so` файл (основно)
- код на C (междинен файл)

Последният да затвори вратата

```
static PyObject *__pyx_f_3fib_fib(PyObject *__pyx_v_n, CYTHON_UNUSED int __pyx_skip_dispatch) {
  PyObject *__pyx_v_a = NULL;
  PyObject *__pyx_v_b = NULL;
  PyObject *__pyx_r = NULL;
  __Pyx_RefNannyDeclarations
  PyObject *__pyx_t_1 = NULL;
  PyObject *__pyx_t_2 = NULL;
  int __pyx_t_3;
  int __pyx_lineno = 0;
  const char *__pyx_filename = NULL;
  int __pyx_clineno = 0;
  __Pyx_RefNannySetupContext("fib", 1);

  __pyx_t_1 = __pyx_int_0;
  __Pyx_INCREF(__pyx_t_1);
  __pyx_t_2 = __pyx_int_1;
  __Pyx_INCREF(__pyx_t_2);
  __pyx_v_a = __pyx_t_1;
  __pyx_t_1 = 0;
  __pyx_v_b = __pyx_t_2;
  __pyx_t_2 = 0;

  while (1) {
    __pyx_t_2 = PyObject_RichCompare(__pyx_v_b, __pyx_v_n, Py_LT); __Pyx_XGOTREF(__pyx_t_2); if (unlikely(!__pyx_t_2)) __PYX_ERR(0, 7, __pyx_L1_error)
    __pyx_t_3 = __Pyx_PyObject_IsTrue(__pyx_t_2); if (unlikely((__pyx_t_3 < 0))) __PYX_ERR(0, 7, __pyx_L1_error)
    __Pyx_DECREF(__pyx_t_2); __pyx_t_2 = 0;
    if (!__pyx_t_3) break;
    __pyx_t_2 = __Pyx_PyObject_CallOneArg(__pyx_builtin_print, __pyx_v_b); if (unlikely(!__pyx_t_2)) __PYX_ERR(0, 8, __pyx_L1_error)
    __Pyx_GOTREF(__pyx_t_2);
    __Pyx_DECREF(__pyx_t_2); __pyx_t_2 = 0;

    __pyx_t_2 = __pyx_v_b;
    __Pyx_INCREF(__pyx_t_2);
    __pyx_t_1 = PyNumber_Add(__pyx_v_a, __pyx_v_b); if (unlikely(!__pyx_t_1)) __PYX_ERR(0, 9, __pyx_L1_error)
    __Pyx_GOTREF(__pyx_t_1);
    __Pyx_DECREF_SET(__pyx_v_a, __pyx_t_2);
    __pyx_t_2 = 0;
    __Pyx_DECREF_SET(__pyx_v_b, __pyx_t_1);
    __pyx_t_1 = 0;
  }
```

Демо

Изводи от междинния код

- Cython генерира доста boilerplate
- Доста от него заради garbage collector-a
  (нещата които включват `INCREF`/`DECREF`)
- Доста заради метаобектния протокол
- Само част от забавянето на Python е заради интерпретатора
  ... и Cython го елиминира
- Останалото е заради MOP (мета-обектния протокол) и GC

Втори опит

```
from libc.stdio cimport printf

def fib(n):
    """Print the Fibonacci series up to n."""
    cdef int a = 0
    cdef int b = 1
    cdef int t
    while b < n:
        printf('%d\n', b)
        t = b
        b = a + b
        a = t
```

Cython vs Python

- Cython синтактично не е надмножество на Python
- ... нито обратното
- ... но синтаксисът има голямо сечение
- Семантиката може да бъде различна
- Има “pure Python mode”, който е по-съвместим
  https://cython.readthedocs.io/en/latest/src/tutorial/pure.html

Има декоратори за performance

ctypes

Ctypes позволява да викаме C функции директно от Python без да ни се налага да пишем и капка C код.

- по-лесният начин
- по-съвместим начин

ctypes основни моменти

- предназначено за комуникация с чист C код -- lean & mean
- стреми се да бъде минимален и да не "пречи", съответно е от сравнително ниско ниво
- зареждане на динамични библиотеки (`.so/.dll`)
- викане на функции от тези библиотеки и даже достъпване на данни
- позволява да описваме интерфейса на функциите, които ще ползваме
- позволява дефиниране на потребителски структури и обединения

Unix vs. Windows

- разликите са основно в начина на зареждане на библиотеките
- `msvcrt.dll` под Windows става на `libc.so.6` (или `libm.so.6`) в Linux

Зареждане на библиотеки

```
>>> from ctypes import *
>>> libc = cdll.LoadLibrary('msvcrt.dll')
>>> libc
<CDLL 'msvcrt.dll', handle 7ffc69b80000 at 6ffffe18ed0>
```

Calling conventions

- `cdll` - `cdecl`
- `windll` - `stdcall`, `oledll` -- само за Windows

Извикване на функции

```
>>> libc.time(None)
1670946095
```

Good morning world

```
>>> libc.printf(b'good morning world'
                b', the time is %d\n', libc.time(None))

good morning world, the time is 1670946095
43
```

ASCII vs UTF-16

```
libc.wprintf('hello world')
libc.printf(b'hello world')
```

Автоматичен marshalling

- `None` → `NULL`
- `int` → `int`
- `bytes` → `const char *`
- `str` → `const wchar_t *`
- останалите трябва да се конвертират

Типове

type	C type	Python type
c_char	char	1-character string
c_wchar	wchar_t	1-character unicode string
c_byte	char	int
c_ubyte	unsigned char	int
c_short	short	int
c_ushort	unsigned short	int
c_int	int	int
c_uint	unsigned int	int
c_long	long	int
c_ulong	unsigned long	int
c_longlong	__int64 or long long	int
c_ulonglong	unsigned __int64 or unsigned long long	int
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	string or None
c_wchar_p	wchar_t * (NUL terminated)	unicode or None
c_void_p	void *	int or None

Обектите от тип c_* са mutable

Имат поле `value`, което може да променяте:

```
>>> i = c_int(42)
>>> print(i, i.value)
c_long(42) 42
>>> i.value = -99
>>> print(i, i.value)
c_long(-99) -99
```

Но Python низовете са immutable!

Низовете са immutable, затова когато използвате функции, които променят аргумента си, трябва да използвате `create_string_buffer`:

```
>>> buf = create_string_buffer(b'hello', 15)
>>> print(sizeof(buf), repr(buf.raw))
15 b'hello\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'
>>> libc.strcat(buf, b', world')
-1214158944
>>> print(sizeof(buf), repr(buf.raw))
15 b'hello, world\x00\x00\x00'
```

Още извикване на функции

```
>>> libc.printf(b'%d bottles of beer\n', 42)
42 bottles of beer
None
>>> libc.printf(b'%f bottles of beer\n', 42.5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 2: <class 'TypeError'>: Don't know how to
convert parameter 2
```

Експлицитно преобразуване

```
libc.printf(b'%f bottles of beer\n', c_double(42.5))
```

```
42.500000 bottles of beer
```

Типове на резулата

Колко е синус от 1?

```
>>> libm.sin(c_double(1))
-1082050016
```

По подразбиране връщаната стойност се интерпретира като int.

```
>>> libm.sin.restype = c_double
>>> libm.sin(c_double(1))
0.8414709848078965
```

Сигнатури

Защото нямаме `.h` файлове.

```
>>> libm.sin.argtypes = [c_double]
>>> libm.sin(1)
0.8414709848078965
```

Аргументи по референция

```
>>> i = c_int()
>>> f = c_float()
>>> s = create_string_buffer('\000' * 32)
>>> print(i.value, f.value, repr(s.value))
0 0.0 b''
>>> libc.sscanf(b'1 3.14 Hello',
...             b'%d %f %s', byref(i), byref(f), s)
3
>>> print(i.value, f.value, repr(s.value))
1 3.1400001049 b'Hello'
```

Структури

```
class POINT(Structure):
    _fields_ = [('x', c_double), ('y', c_double)]

>>> point = POINT(10, 20)
>>> print(point.x, point.y)
10.0 20.0
>>> point = POINT(y=5)
>>> print(point.x, point.y)
0.0 5.0
>>> POINT(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: too many initializers
```

Структури (2)

Структурите имат метаклас, различен от стандартния `type`:

```
>>> type(POINT) == type
False
>>> type(POINT)
<class '_ctypes.PyCStructType'>
```

Влагане на структури

```
class RECT(Structure):
    _fields_ = [('upperleft', POINT), ('lowerright', POINT)]
```

```
>>> rc = RECT(point)
>>> print(rc.upperleft.x, rc.upperleft.y)
0.0 5.0
>>> print(rc.lowerright.x, rc.lowerright.y)
0.0 0.0
>>> rc = RECT((1, 2), (3, 4))
>>> rc = RECT(POINT(1, 2), POINT(3, 4))
```

Подравняване и byte-order

Две думи -- подразбира native

- за повече виж документацията

Масиви

Масивите имат тип, който включва броя елементи. Получаваме го като умножим типа на елементите по техния брой -- (`ElementType * element_count`). Получаваме нов тип:

```
POINT_ARRAY_10 = POINT * 10
for i in POINT_ARRAY_10():
    print(i.x, i. y)

>>> (c_int * 3)(1,2,3)[0]
1
```

Указатели

- всеки път създава нов `c_*` обект
- `p.contents is p.contents == False`
- но реално C обектите, които те представят, са един и същи

Указатели (2)

```
>>> i = c_int(10)
>>> p = pointer(i)
>>> p.contents
c_int(10)
>>> j = c_int(10)
>>> p.contents = j
>>> p.contents.value
10
>>> p.contents.value = 11
>>> j.value
11
>>> p[0]
11
```

Тип на указателите

Указателите към c_* обекти си имат собствен тип:

```
>>> point_p = pointer(point)
>>> type(point_p)
<class '__main__.LP_POINT'>
>>> type(point_p) == POINTER(POINT)
True
>>> POINTER(POINT)
<class '__main__.LP_POINT'>
```

POINTER(c_char) vs c_char_p

- POINTER се използва за указатели към `c_*` обекти
- `c_{char,wchar,void}_p` са указатели към C обекти

Когато функция връща `(char *)`, съответния `.restype` атрибут трябва да бъде `c_char_p`...

POINTER(c_char) vs c_char_p (2)

```
>>> libc.strstr.restype = POINTER(c_char)
>>> found_p = libc.strstr(b'abc def ghi', b'def')
>>> found_p[:5]
b'\x00\x00\x00\x00\x00' # нищо общо
>>> libc.strstr.restype = c_char_p # strstr връща указател към C памет,
...                                 # а не към питонски c_char
>>> found_p = libc.strstr(b'abc def ghi', b'def')
>>> found_p
'def ghi'
```

`ctypes` прави автоматично преобразуване, създавайки нов `str` обект

Преобразуване на масиви към указатели

Ctypes е стриктен и рядко прави преобразувания. Затова ни се налага ние да ги правим, използвайки функцията `cast`:

```
>>> libc.strstr.argtypes = [c_char_p, c_char_p]
>>> byte_array = (c_byte * 12)(*b'abc def ghi')
>>> byte_array[:]
[97, 98, 99, 32, 100, 101, 102, 32, 103, 104, 105, 0]
>>> libc.strstr(byte_array, b'def')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ctypes.ArgumentError: argument 1: <class 'TypeError'>: wrong type
>>> libc.strstr(cast(byte_array, c_char_p), b'def')
'def ghi'
```

Указатели

```
>>> p[10]
159787148
>>> p[10] = 20
>>> p[10]
20
```

Ctypes дълбоки води

- вложени структури
- callbacks
- byte ordering
- аргументи по референция
- Задължително погледнете за изненадите. (в документацията)

Ctypes pros/cons

+ работи за Linux, Mac OS X и Windows
+ работи между версии на CPython
+ работи за алтернативни имплементации на Python
+ по-лесно и безболезнено отколкото Пайтън C API
+ особено когато комуникираме с чист C код
- от ниско ниво е
- трябва да пишем доста Пайтън код
- трябва да се грижим за marshalling
- почти невъзможно да викаме C++ код

Алтернативи на ctypes / C API

- SWIG & Boost.Python
- Трябва да работите със source файлове
- Автоматичен marshalling, който всъщност... работи
- Човечна C++ поддръжка

Въпроси?