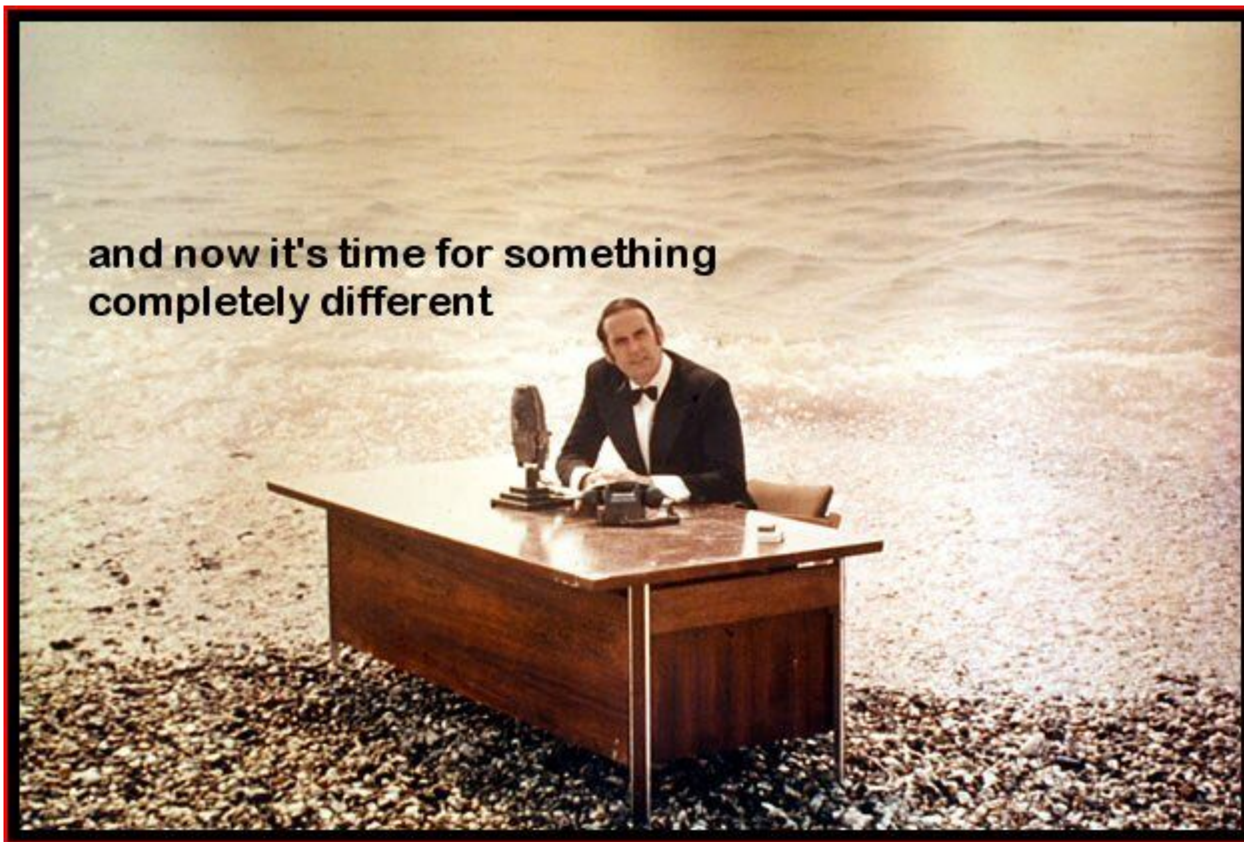
24. ctypes

14 януари 2025


and now it's time for something
completely different

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

ctype	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?