21. Python отвътре

07 януари 2025

Преговор - модули

Какво може да бъде модул в python?

- Файл с разширение `.py`
- Директория съдържаща файл с име `__init__.py`
- Можем да ги видим с `__file__` атрибута

```
>>> import functools
>>> functools.__file__
'C:\\dev\\Python311\\Lib\\functools.py'
```

Ho!

```
>>> import pyexpat
>>> pyexpat.__file__
'C:\\dev\\Python311\\DLLs\\pyexpat.pyd'
```

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>
```

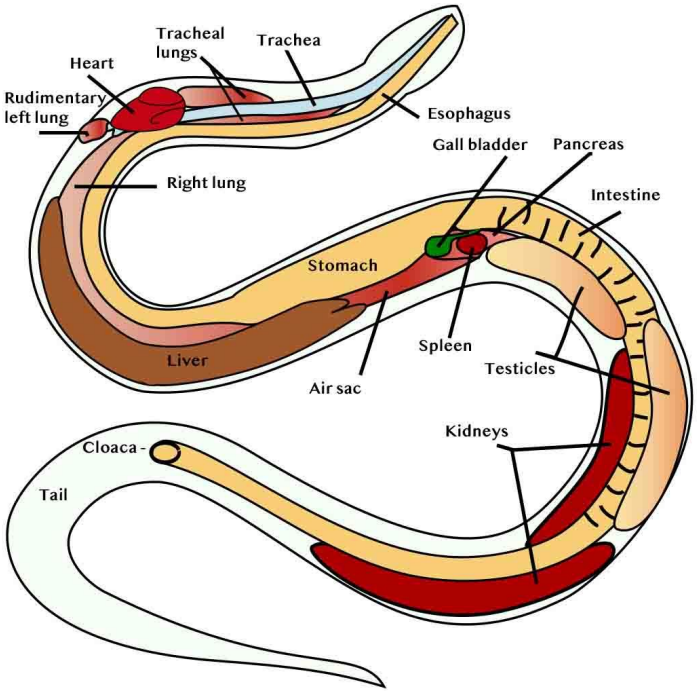
Ho!

```
>>> 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

Какво знаем за Python?



Python

- интерпретиран
- динамичен
- ядро написано на C
- стандартна библиотека на Python
- ... и малко C

Интерпретатор?

- Interpreter vs translator
  (симултантен превод / превод)
- Алтернативи на интерпретаторите?

Таксономия

- компилатори
- JIT компилатори
- интерпретатори

Компилирани езици

Популярни примери:

- C
- C++
- Rust

Компилатори (подготовка)

- синтактичен анализ
- оптимизация
- транслация към изпълним код
  - изпълними файлове (`.exe`, `a.out`)
  - библиотеки (`.dll`, `.so`)
  - формати (PE/ELF/COFF/Mach-O)

Компилатори (изпълнение)

- зарежда се изпълним файл
- зареждат се библиотеки
- обработват се секциите в тях (данни, код, ресурси)
- процесорът изпълнява машинния код

Какво е машинен код?

- assembler?
- нули и единици?
- opcodes

JIT компилирани езици

Популярни примери:

- C#
- Java

JIT компилатори (подготовка)

- синтактичен анализ
- транслация към байткод
  - `.jar` (Java)
  - `.dll` (C#; тези dll-и са малко по-различни)

JIT компилатори (изпълнение)

- зарежда се виртуална машина (runtime)
  - виртуалната машина обикновено е native code
  - “native code” е просто машинен код
- зарежда се байткод
- при първо извикване байткодът се компилира
- при нужда - profiling / optimization

Какво е байткод

- opcodes
- инструкции (като асемблер)
- нули и единици
- подобна идея като машинния код
- ... но за виртуален процесор

Интерпретирани езици

- Python
- Ruby
- JavaScript
- Perl

Интерпретатори (подготовка)

???

Интерпретатори (подготовка)

... кодът си остава просто текст

Интерпретатори (изпълнение)

- зарежда се интерпретатор
  - обикновено е native code
- зарежда се сорс код
- синтактичен анализ
- вътрешна (in-memory) репрезентация
  - AST
  - bytecode
- Вътрешната репрезентация се изпълнява

Конкретно за Python

- Има интерпретиран байткод
- стандартна библиотека (на Python)
  - `.py`
  - `.pyc`
- `python.dll`
  - builtin типове
  - builtin функции
- можем да си напишем наши компилирани builtins
  - `.pyd` (Windows)
  - `.so` (Linux)

CPython

- Името на официалната имплементация
- Нещата в предния слайд са имплементационни детайли!
- Важат за CPython 3.12
- CPython 3.13 вече има експериметален JIT
- ... в някакъв момент в бъдещето JIT ще бъде default-а

Python имплементации

Различни питони:

- CPython # интерпретиран
- IronPython # JIT върху .NET
- Jython # JIT върху JVM
- Pyston # JIT чрез LLVM; unmaintained
- Unladen Swallow # JIT; dead
- Cython # компилиран (с уговорки)

JavaScript имплементации

Python не е специален!

JavaScript също има множество имплементации:

- V8 (JIT)
- Chakra
- JavaScriptCore
- SpiderMonkey
- Rhino (JIT върху Java)

Java имплементациии

- OpenJDK:
  - HotSpot – JIT компилатор
  - GraalVM – JIT компилатор/интерпретатор/AOT (ahead-of-time) компилатор
  - Project Zero – JIT компилатор
  - байткод интерпретатор
- Eclipse OpenJ9 – JIT компилатор/AOT (ahead-of-time) компилатор
- GCJ – GNU Compiler for the Java; inactive
- JOP – хардуер; inactive
- … и още много други

Обратно към CPython

- `.py` е сорс код на Python
- `.pyc` е кеширан байткод
- Във Windows
  `%PYTHONPATH%`
- Във Unix (и приятели)
  `/usr/lib/python*/**`
  `/usr/bin/python`

... нека разгледаме малко Python байткод!

Примерна функция

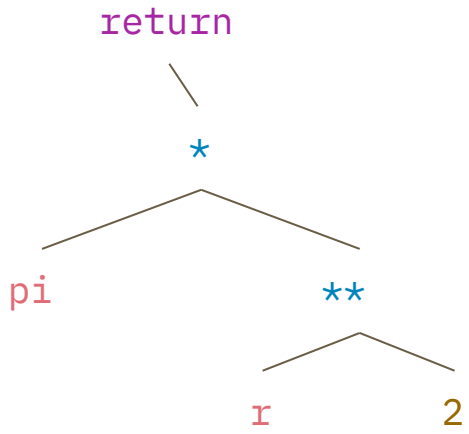
```
from math import pi

def circle_area(r):
    return pi * (r ** 2)

print(circle_area.__code__.co_code)
# b't\x00j\x01|\x00d\x01\x13\x00\x14\x00S\x00'
```

import dis

```
>>> import dis
>>> dis.dis(circle_area.__code__.co_code)
          0 LOAD_GLOBAL              0 (0)
          4 LOAD_FAST                0 (0)
          6 LOAD_CONST               1 (1)
          8 BINARY_POWER
         10 BINARY_MULTIPLY
         12 RETURN_VALUE
```

Абстрактно синтактично дърво (AST)

- Дървовидна репрезентация на програма
- Тялото на `circle_area` (грубо) изглежда така:



Инфиксен запис

- Обхождаме AST в дълбочина
- Обхождаме в ред ляво-корен-дясно
- Получаваме:
  ```
  (return (pi * (r ** 2)))
  ```
- **Забележка 1:** ако знаем приоритетите не са нужни скоби
- **Забележка 2:** като “нормален” език за програмиране

Префиксен (полски) запис

- Обхождаме AST в дълбочина
- Обхождаме в ред корен-ляво-дясно
- Получаваме:
  `(return (* pi (** r 2)))`
- **Забележка 1:** Това е валиден Lisp (Scheme) код:
  `(* pi (expt r 2))`
- **Забележка 2:** ако знаем арността не са нужни скоби

Суфиксен (обратен полски) запис

- Обхождаме AST в дълбочина
- Обхождаме в ред ляво-дясно-корен
- Получаваме:
  ```
  ((pi (r 2 **) *) return)
  ```
- **Забележка 1:** ако знаем арността не са нужни скоби
- **Забележка 2**: това е валиден PostScript код:
  ```
  PI r 2 exp mul
  ```

Байткод!

```
((pi (r 2 **) *) return) - със скоби
pi  r  2  **  *  return - без скоби
```

```
LOAD_GLOBAL 0          ⇔   pi
LOAD_FAST 0       ⇔   r
LOAD_CONST   1         ⇔   2
BINARY_POWER      ⇔   **
BINARY_MULTIPLY   ⇔   *
RETURN_VALUE      ⇔   return
```

Байткод (2)

- Python байкода е обратен полски запис на AST-то
- ... подобно на JVM (Java)
- ... подобно на .NET (C#)
- ... подобно на p-code (Pascal)

import ast

Модул, с който да сглобяваме AST/код

builtins

- Горното важи за функции написани на Python
- Както споменахме – не всичко в Python е написано на Python

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)
- shared object/dynamic link library са два термина за едно и също нещо
- представляват C библиотека
- ... обикновено в комбинация с `.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;
  }
```

JIT компилатори/интерпретатори

- разграничението често е условно
- ... и зависи от имплементацията
- на теория: теория == практика
- на практика: теория != практика

Въпроси?