

Python

Семинар 1

Преподаватель: Дмитрий Косицин
BSU FAMCS (Fall'19)

Модули и пакеты

...

Схема импорта модулей

При вызове «import x» происходит следующее:

1. find module
2. load module # an object loaded_module is created
 - create an object type of ModuleType
 - read source
 - compile source
 - execute source
3. sys.modules['x'] = loaded_module
4. <this_module>.x = loaded_module

Загрузка и перезагрузка модуля

reload(loaded_module) – перезагружает модуль, не создавая новый объект.

Сравните:

```
>>> from module import x  
>>> # then use x directly
```

```
>>> import module  
>>> # then use module.x
```

Важно! Это не то же самое, что удалить модуль и заново загрузить. В случае **reload** (`importlib.reload` в Python 3) все объекты в модуле *пересоздаются!* Более того, **reload** имеет множество подводных камней.

Загрузка модулей

При загрузке создаются и вызываются default-значения функций (вопрос: что произойдет?):

```
>>> def f (x=g ()) :  
>>>     pass
```

```
>>> def g () :  
>>>     return 0
```

Для изменения поведения при исполнении модуля от поведения при импорте используется следующее:

```
>>> if __name__ == '__main__':  
>>>     # code to be executed if module is an entry point
```

Импорт модулей

Модуль можно загружать по имени:

```
>>> import importlib
```

```
>>> module_instance = importlib.import_module('module_name')
```

Есть возможность загружать source, compiled (.pyc) и dynamic (.pyd, .so) модули по полному пути (см. *imp* в Python 2 и *importlib.util* в Python 3)

Очередность загрузки:

- package
- module
- namespace ([PEP 420](#), Python 3.3+)

Пакеты и пространства имен

Package – папка с модулями, где присутствует файл `__init__.py`.

Namespace – файл `__init__.py` отсутствует.

Разница будет для вложенных папок: package вложенный обнаружится, а для namespace нужно явно прописать путь в **sys.path**.

Напоминание. `sys.path.append(x)` равносильно `sys.path += x`, но отличается от `sys.path = sys.path + x`

Замечание. Узнать имя файла из модуля можно обратившись к переменной `__file__`, имя модуля – к `__name__`.

ДОПОЛНИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Модули можно загружать прямо из *zip*-архива.

Можно использовать относительные импорты ([PEP-328](#)).

Для **moduleX.py** верны следующие относительные импорты:

```
package/  
  __init__.py  
  subpackage1/  
    __init__.py  
    moduleX.py  
    moduleY.py  
  subpackage2/  
    __init__.py  
    moduleZ.py  
  moduleA.py
```

```
from .moduleY import spam  
from .moduleY import spam as ham  
from . import moduleY  
from ..subpackage1 import moduleY  
from ..subpackage2.moduleZ import eggs  
from ..moduleA import foo  
from ...package import bar  
from ...sys import path
```


Дополнительные возможности.

Замечания

Относительные импорты не столь распространены ввиду худшей переносимости.

У модуля также может присутствовать *docstring* – его следует располагать вверху файла в тройных кавычках.

Модуль `__future__` является директивой компилятору создать `.pyc` файл, используя другие инструкции.

Существуют дополнительные механизмы: `path hooks`, `metapath`, `module finders and loaders`, etc.

Логирование

...

Логирование

```
import sys
import logging
import datetime

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)
logger.addHandler(logging.handlers.StreamHandler(sys.stdout))

logger.info("current year's %d", datetime.datetime.today().year)
```

Замечание. В Python 3 Handlers и Filters расположены во вложенных модулях, а содержимое модуля datetime перемещено.

Логирование: подробности

Объекты **Logger** объявлены в модуле **logging**.

Создать новый logger можно вызовом **getLogger**, передав ему некоторое имя.

Замечание. Если передать имя существующего логгера в метод **getLogger**, то будет возвращен уже существующий логгер с таким именем.

Логировать сообщение можно с помощью методов **debug**, **info**, **warning**, **error**, **critical** или общего **log**.

Установить уровень чувствительности (verbosity) можно методов **setLevel**.

Логирование: подробности

Добавить обработчик можно методом **addHandler**, фильтр – **addFilter**.

Реализованные обработчики: **StreamHandler**, **FileHandler**, **RotatingFileHandler**, **SocketHandler**, etc.

Замечание. Все handler'ы и filterer'ы имеют базовые классы - **logging.Handler** и **logging.Filterer**.

Конфигурация логгера может быть сохранена в файле и загружена с помощью **logging.config.dictConfig**.

Форматирование строк

...

Форматирование строк

Поддерживается *printf*-style форматирование:

```
>>> 'number of %.3f values in %s is %d' % (0.1234, 'some object', 3)
number of 0.123 values in some object is 3
```

Есть возможность использовать именованные аргументы:

```
>>> 'number of %(name)s is %(count)d' % {'name': 'names', 'count': 2}
number of names is 2
```

Поддерживается новый стиль форматирования строк:

```
>>> 'number of {0:.3f} values in {1} is {2}'.format(
    0.1234, 'some object', 3)
number of 0.123 values in some object is 3
```

Форматирование строк

А еще можно:

- аналогично использовать именованные аргументы: `" { name } "`
- индексировать аргументы: `" { items [0] } "`
- обращаться к атрибутам: `" { point . x } "`
- опускать индексы: `" { } { } "`
- повторять и менять местами индексы: `" { 1 } { 0 } { 1 } "`

Вопрос: что будет, если не совпадает количество аргументов для подстановки? А если нету такого именованного аргумента?

Форматирование строк

Замечание. Если вам нужно подставить множество локальных переменных, можно использовать словари **locals()** и **globals()**, определенные интерпретатором:

```
>>> x = 2
>>> "{x}".format(**locals())
2
```

Вопрос: верно ли, что так хитро можно менять значения локальных переменных?

Замечание. В Python 3 добавлен метод **format_map**, чтобы передавать словарь не распаковывая.

Работа с командной строкой

...

Парсинг аргументов командной строки

```
import argparse

parser = argparse.ArgumentParser(description='Process some
integers.')

parser.add_argument('integers', type=int, nargs='+',
                    help='an integer for the accumulator')
parser.add_argument('--sum', dest='accumulate',
                    action='store_const',
                    const=sum, default=max,
                    help='sum the integers (default: find the
max) ')

args = parser.parse_args()
print(args.accumulate(args.integers))
```

Для парсинга есть **ArgumentParser** в модуле **argparse**.

Методы **parse_args** и **parse_known_args** принимают некоторый список аргументов (по умолчанию **sys.argv**), парсят его и возвращают объект **Namespace**.

Произвольную строку запуска можно разбить на список с помощью модуля **shlex**.

Информации об аргументе добавляется с помощью **add_argument**:

- имена переменных через '-' (dash), '--' (double dash) или без них
- *dest* – имя переменной, в которой хранится значение
- *type* – преобразование типа
- *action* – действие при получении аргумента (*store*, *store_true*, *append*, etc.)
- *nargs* – количество аргументов (1, 1 и более, 0 и более)
- *default* – значение по умолчанию (если не передан)
- *required* – обязательный аргумент
- *choices* – список возможных значений
- *help* – описание аргумента

Модули стандартной библиотеки

...

Работа с файловой системой. Регулярные выражения. Модуль `functools`.

Работа с файловой системой

В стандартной библиотеке есть несколько модулей, отвечающих за файловую систему.

Модуль **os.path** служит, в основном, для работы с путями:

**join | abspath | relpath | commonprefix | split | normpath
walk | getsize | exists | isfile**

Для работы с файловой системой используется модуль **os**:

listdir | mkdir | makedirs | remove | rmdir | rename | stat

Работа с файловой системой

Для копирования и удаления файлов используется модуль **shutil**:

copy | move | rmtree

Модули **glob** и **fnmatch** предназначены для поиска файлов и папок по шаблонам с wildcard'ами, для сравнения файлов есть модуль **filecmp**.

В Python 3.6 появились полноценная библиотека для работы одновременно с путями и файловыми объектами – Pathlib ([PEP-428](#)), а также функция `os.scandir` – улучшенный аналог `os.walk` ([PEP-519](#)).

Работа с системой

Модуль **os** также содержит множество констант и функций для работы с системой:

chdir | **getcwd** | **getenv** | **putenv** | **unsetenv**
environ | **extsep**

Все функции являются *system specific* и могут отсутствовать. Для управления процессами есть **abort** и **kill**.

В Python 3 был систематизирован весь протокол работы с файлами / стримами – см. модуль **io**.

Полезным в модуле **io** может быть файловый буффер **StringIO** (в Py2 отдельно).

Модули `re` и `functools`

В модуле `re` собраны функции для работы с регулярными выражениями:

- **`search` / `match` / `finditer`** – поиск шаблона, возвращают **`MatchObject`**
- **`split` / `sub`** – разбиение строки по шаблону / замена подстроки

`MatchObject` имеет следующие свойства: **`groups`, `groupdict`, `start`, `end`, `span` и `pos`.**

В модуле **`functools`** содержатся полезные функции:

- ***`partial`*** – возвращает новую функцию, фиксируя некоторые аргументы (замена, например **`lambda x: f(x, True)`**; также ***`partialmethod`*** в Py3.4+)
- ***`lru_cache`*** – декоратор для кэширования результатов функции в LRU-кэше (Py3.2+)
- ***`singledispatch`*** – декоратор, позволяющий вызывать функцию в зависимости от типа ее аргумента (Py3.4+, [примеры](#))

Интерпретатор. Байткод

Для Python кода в стандартной библиотеке есть AST-парсер.

Код можно либо преобразовать в Abstract Syntax Tree ([ast module](#)), либо скомпилировать – преобразовать в байткод, который далее интерпретировать ([dis module](#)).

Каждой инструкции байткода соответствует функция в интерпретаторе, которая ее выполняет.

Замечание. Подробнее о байткоде тут: [article](#), [article](#).

Замечание. Статья об интерпретаторе тут: [article](#), [slides](#).

Пример байткода

```
import dis

global_a = 0

def f(closure_b):
    enclosing_c = 1

    def g(param_d):
        local_e = 2
        return (global_a +
                closure_b + enclosing_c +
                param_d + local_e)

    dis.dis(g)

f(-1)
```

8	0	LOAD_CONST	1	(2)
	3	STORE_FAST	1	(local_e)
9	6	LOAD_GLOBAL	0	(global_a)
	9	LOAD_DEREF	0	(closure_b)
	12	BINARY_ADD		
	13	LOAD_DEREF	1	(enclosing_c)
	16	BINARY_ADD		
	17	LOAD_FAST	0	(param_d)
	20	BINARY_ADD		
	21	LOAD_FAST	1	(local_e)
	24	BINARY_ADD		
	25	RETURN_VALUE		

Интерпретаторы

Есть множество реализаций интерпретаторов, которые написаны на разных языках программирования:

- CPython, Jython, IronPython – интерпретаторы на C, Java и .Net
- PyPy, Numba – Just-in-Time compilers
- Cython, Nuitka – использование типов и оптимизации (на базе CPython)
- Stackless Python, Julia – прочие реализации и расширения языка

Естественно, что если интерпретатор написан на C, то можно из Python напрямую взаимодействовать с C-кодом – реализовывать C/C++ extensions ([standard library](#), [boost](#), [Pybind11](#)).

Интерпретатор CPython

Все объекты в CPython описываются структурами.

Типу **object** соответствует структура **PyObject**, с указателем на которую он повсеместно работает.

Список в Python – аналог **vector** в C++ STL, расширяется в 9/8 раз (плюс константа; см. *listobject.c*).

Словарь в Python – хэш-таблица с открытой адресацией (см. *dictobject.c*):

- минимальный размер по умолчанию 8
- смещение считается как $j = ((5*j) + 1) \bmod 2^{*i}$
- расширяется при наполненности от 1/2 до 2/3 в 2-3 раза от количества хранимых элементов

GIL and GC

В Python есть **Global Interpreter Lock** – механизм, который гарантирует одновременное выполнение только одного потока. Переключение **GIL** в последней версии Python происходит по таймеру.

Для всех объектов в Python ведется счетчик ссылок, а все объекты классифицируются в три поколения. **GC** также умеет разрешать циклические зависимости, если у объектов не переопределен метод `__del__`.

Ответы и полезные ссылки

...

ОТВЕТЫ

Строки модуля интерпретируются последовательно. При попытке создания объекта **f** – функции аргументы по умолчанию будут также интерпретированы и сохранены в данном объекте. Поскольку функция **g** объявлена ниже, произойдет исключение **NameError**, что такого имени нету. Обратите внимание, что значения аргументов по умолчанию создаются *только один раз* при загрузке модуля.

ОТВЕТЫ

Форматирование строк

Если количество аргументов для подстановки не совпадает с количеством в шаблоне или один из требуемых именованных аргументов не передан, произойдет исключение **TypeError**.

Однако для подстановки можно передать словарь, в котором значений больше, чем требуется. Ошибки в таком случае не будет.

ПОЛЕЗНЫЕ ССЫЛКИ

Множество *shortcuts* можно найти в книге Pilgrim, М. Dive Into Python 3.

Подробнее механизм импортов описан здесь (Python 3):

- <https://docs.python.org/3.7/library/modules.html>
- <https://docs.python.org/3.7/tutorial/modules.html>
- <https://docs.python.org/3.7/reference/import.html>

Для Python 2 документация расположена по следующим ссылкам:

- <https://docs.python.org/2.7/library/modules.html>
- <https://docs.python.org/2.7/tutorial/modules.html>