Программирование на Python

лекция 4

Владимир Юрьевич Полищук

Инженерная школа
информационных технологий и
робототехники,
Томский политехнический университет

pvy3@tpu.ru

Содержание

- Функции
- Модули

Основы функций

Функция – это средство, позволяющее группировать наборы инструкций так, что в программе они могут запускаться неоднократно.

Функции обеспечивают *многократное использование* программного кода и уменьшающие его *избыточность*.

Зачем нужны функции?

- *Максимизировать многократное использование программного кода и минимизировать его избыточность*
- *Процедурная декомпозиция*

Создание функций

Основные концепции, составляющие основу функций в языке Python:

- ***def*** – это исполняемый программный код. Функции в языке Python создаются с помощью новой инструкции **def** (создает объект и присваивает ему имя).
- Выражение ***lambda*** создает объект и возвращает его в виде результата.
- ***return*** передает объект результата вызывающей программе.
- **yield** передает объект результата вызывающей программе и запоминает, где был произведен возврат.
- аргументы передаются посредством присваивания (в виде ссылок на объекты).

Создание функций (2)

Основные концепции, составляющие основу функций в языке Python:

- **global** объявляет глобальные переменные для модуля, без присваивания им значений.
- **nonlocal** объявляет переменные, находящиеся в области видимости объемлющей функции, без присваивания им значений.
- Аргументы получают свои значения (ссылки на объекты) в результате выполнения операции присваивания.
- Аргументы, возвращаемые значения и переменные не объявляются.

Инструкция def

def <name>(arg1, arg2,... argN): **# возвращает объект *None***
    <statements>

def <name>(arg1, arg2,... argN):
        ...
    return <value>

def <name>(arg1, arg2,... argN):
        ...
    yield <value>

Инструкция def

```
if test:
        def func():  # Определяет функцию таким способом
        ...
else:
        def func():  # Или таким способом
        ...
...
func() # Вызов выбранной версии

othername = func  # Связывание объекта функции с
                  именем
othername() # Вызов функции

def func():...    # Создает объект функции
func()            # Вызывает объект
func.attr=value # Присоединяет атрибут к объекту
```

!!

Функции имеют две стороны:

- определение (инструкция def, которая создает функцию)
- вызов (выражение, которое предписывает интерпретатору выполнить тело функции).

Определение

```
def times(x, y): # Создать функцию и связать ее с
                 # именем
    return x * y # Тело, выполняемое при вызове
                 # функции
```

Вызов

```
>>> times(2, 4)  # Аргументы в круглых скобках
8
```

```
>>> x = times(3.14, 4) # Сохранить объект
                       результата
>>> x
12.56

>>>times('word', 2)  # Функции не имеют типа
'wordword'
```

Полиморфизм в языке Python

```
def times(x, y):
        return x * y
```

полиморфизм – термин, который означает, что смысл операции зависит от типов обрабатываемых объектов.

!!

- Оформив программный код в виде функции, появляется возможность использовать его столько раз, сколько потребуется.
- Так как вызывающая программа может передавать функции произвольные аргументы, функция сможет использоваться с любыми двумя последовательностями (или итерируемыми объектами) для получения их пересечения.
- Когда логика работы оформлена в виде функции, достаточно изменить программный код всего в одном месте, чтобы изменить способ получения пересечения.
- Поместив функцию в файл модуля, ее можно будет импортировать и использовать в любой программе на вашем компьютере.

Пример

```
def intersect(seq1, seq2):
    res= [] # Изначально пустой результат
    for x in seq1: # Обход последовательности seq1
        if x in seq2: # Общий элемент?
            res.append(x) # Добавить в конец
    return res
```

```
>>> s1 = “SPAM
>>> s2 = “SCAM”
>>>intersect(s1, s2)   # Строки
[‘S’, ‘A’, ‘M’]

>>> [x for x in s1 if x in s2] # генератор списков
[‘S’, ‘A’, ‘M’]
```

Еще о полиморфизме

```
>>>x = intersect([1, 2, 3], (1, 4)) # Смешивание
                                    типов
>>>x   # Объект с результатом
[1]
```

Локальные переменные

```
def intersect(seq1, seq2):
    res= [] # Изначально пустой результат
    for x in seq1: # Обход последовательности seq1
        if x in seq2: # Общий элемент?
            res.append(x) # Добавить в конец
    return res
```

Почти все переменные в функции *intersect* являются локальными, так:

- Переменная ***res*** явно участвует в операции присваивания, поэтому она – локальная переменная.
- Аргументы передаются через операцию присваивания, поэтому ***seq1*** и *seq2* тоже локальные переменные.
- Цикл ***for*** присваивает элементы переменной, поэтому имя ***x*** также является локальным.

Области видимости в языке Python

• Имена, определяемые внутри инструкции ***def***, видны только программному коду внутри инструкции ***def***. К этим именам нельзя обратиться за пределами функции.

• Имена, определяемые внутри инструкции ***def***, не вступают в конфликт с именами, находящимися за пределами инструкции ***def***, даже если и там и там присутствуют одинаковые имена.

Области видимости в языке Python

- Если присваивание переменной выполняется внутри инструкции ***def***, переменная является **локальной** для этой функции.
- Если присваивание производится в пределах объемлющей инструкции ***def***, переменная является **нелокальной** для этой функции.
- Если присваивание производится за пределами всех инструкций ***def***, она является **глобальной** для всего файла.

```
X = 99
def func():
    X = 88
```

Правила видимости имен

• **Объемлющий модуль** – это глобальная область видимости.

• **Глобальная область** видимости охватывает единственный файл. Не надо заблуждаться насчет слова «глобальный» – имена на верхнем уровне файла являются глобальными только для программного кода в этом файле.

• Каждый вызов функции создает новую **локальную область** видимости. Всякий раз, когда вызывается функция, создается новая локальная область видимости – то есть пространство имен, в котором находятся имена, определяемые внутри функции.

• Операция присваивания создает локальные имена, если они не были объявлены глобальными или нелокальными. global , nonlocal

• Все остальные имена являются локальными в области видимости объемлющей функции, глобальными или встроенными.

!!

```
L = []
L.append(X)
L =X
```

Разрешение имен: правило LEGB

Для инструкции *def*:

• Поиск имен ведется самое большее в четырех областях видимости: локальной, затем в объемлющей функции (если таковая имеется), затем в глобальной и, наконец, во встроенной.

• По умолчанию операция присваивания создает локальные имена.

• Объявления *global* и *nonlocal* отображают имена на область видимости вмещающего модуля и функции соответственно.

Разрешение имен: правило LEGB

Когда внутри функции выполняется обращение к неизвестному имени, интерпретатор пытается отыскать его в четырех областях видимости:

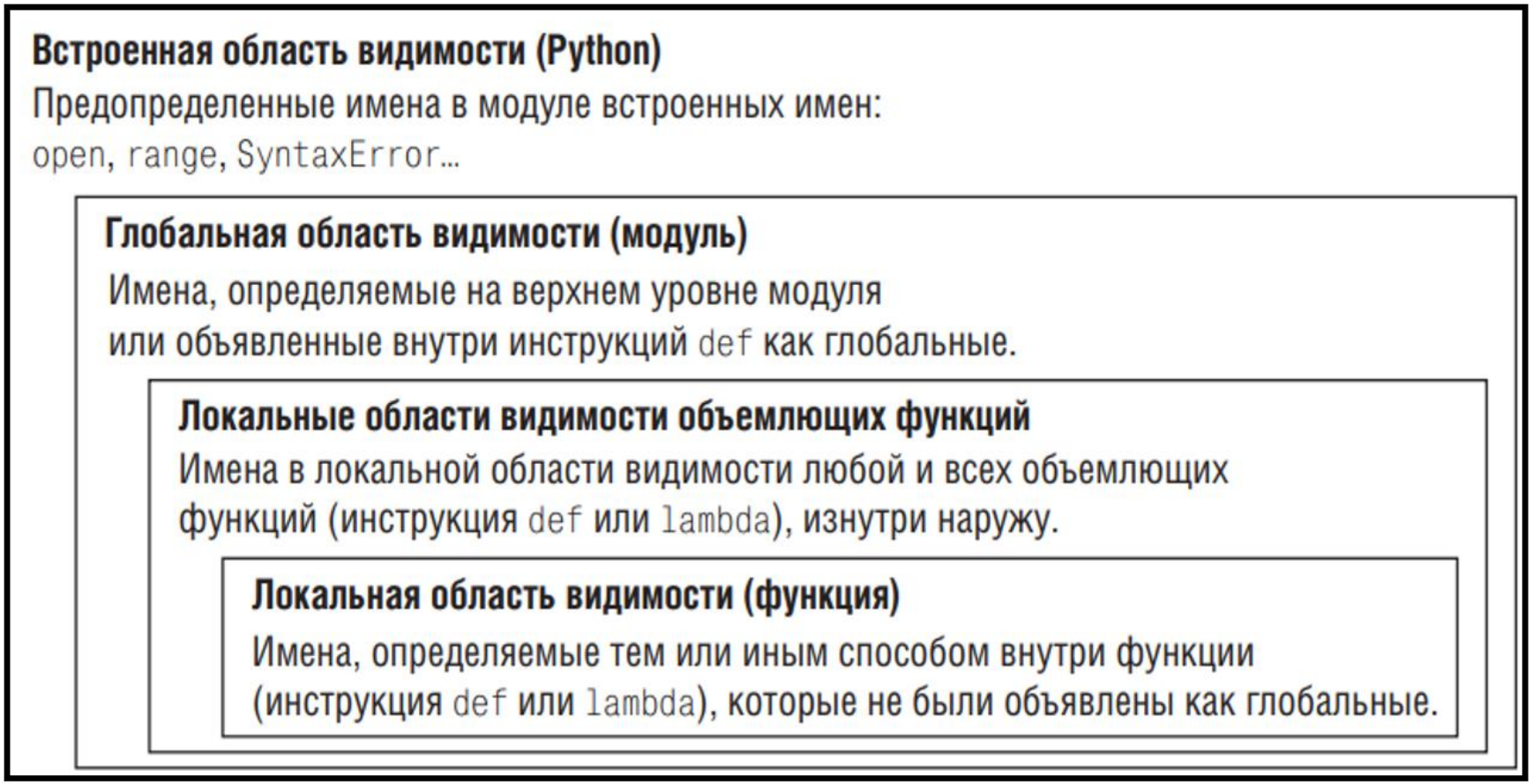
- в **локальной (local, L)**,
- затем в локальной области любой объемлющей инструкции ***def*** **(enclosing, E**) или в выражении *lambda*,
- затем в глобальной (**global, G**)
- и, наконец, во встроенной (**built-in, B)**.

Поиск завершается, как только будет найдено первое подходящее имя.

Разрешение имен: правило LEGB

- Когда внутри функции выполняется операция присваивания, интерпретатор всегда создает или изменяет имя в локальной области видимости, если в этой функции оно не было объявлено глобальным или нелокальным.
- Когда выполняется присваивание имени за пределами функции (то есть на уровне модуля или в интерактивной оболочке), локальная область видимости совпадает с глобальной – с пространством имен модуля.

Разрешение имен: правило LEGB



Пример области видимости

```
# Глобальная область видимости
X = 99 # X и func определены в модуле: глобальная область
def func(Y): # Y и Z определены в функции: локальная область
        # Локальная область видимости
        Z=X+Y  # X – глобальная переменная
        return Z
func(1) # func в модуле: вернет число 100
```

Встроенная область видимости

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException', 'BlockingIOError',
'BrokenPipeError', 'BufferError', 'BytesWarning', 'ChildProcessError', 'ConnectionAbortedError',
'ConnectionError', 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FileExistsError', 'FileNotFoundError',
'FloatingPointError', 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'InterruptedError', 'IsADirectoryError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'ModuleNotFoundError', 'NameError', 'None',
'NotADirectoryError', 'NotImplemented', 'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError', 'RecursionError',
'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning', 'StopAsyncIteration',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'TimeoutError',
'True', 'TypeError', 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'WindowsError', 'ZeroDivisionError', '__build_class__', '__debug__', '__doc__', '__import__',
'__loader__', '__name__', '__package__', '__spec__', 'abs', 'all', 'any', 'ascii', 'bin', 'bool',
'breakpoint', 'bytearray', 'bytes', 'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit', 'filter', 'float', 'format',
'frozenset', 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct', 'open',
'ord', 'pow', 'print', 'property', 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

Встроенная область видимости

```
>>>zip # Обычный способ
        <class zip>
>>>import builtins  # Более сложный способ
>>>builtins.zip
        <class zip>

 def hider():
     open= 'spam'   # Локальная переменная,
                      переопределяет встроенное имя
       ...
     open('data.txt')  # В этой области видимости файл
                         не будет открыт!
```

Встроенная область видимости

```
X = 88 # Глобальная переменная X

def func():
    X = 99 # Локальная переменная X:
           # переопределяет глобальную

func()
print(X) # Выведет 88: значение не изменилось
```

Инструкция global

Инструкция ***global*** сообщает интерпретатору, что функция будет изменять одно или более глобальных имен, то есть имен, которые находятся в области видимости (в пространстве имен) вмещающего модуля.

Инструкция global

Общая информация об инструкции *global*:

• Глобальные имена – это имена, которые определены на верхнем уровне вмещающего модуля.

• Глобальные имена должны объявляться, только если им будут присваиваться значения внутри функций.

• Обращаться к глобальным именам внутри функций можно и без объявления их глобальными.

Инструкция global

```
X = 88 # Глобальная переменная X
def func():
    global X
    X = 99 # Глобальная переменная X: за
           пределами инструкции def
func()
print(X) # Выведет 99

y,z= 1, 2  # Глобальные переменные в модуле
def all_global():
    global x  # Объявляется глобальной для
                присваивания
    x=y+z  # Объявлять y, z не требуется: применяется
             правило LEGB
```

Минимизируйте количество глобальных переменных

```
X = 99
def func1():
    global X
    X = 88
def func2():
    global X
    X = 77
```

Передача аргументов

- Аргументы передаются через автоматическое присваивание объектов локальным переменным.
- Операция присваивания именам аргументов внутри функции не оказывает влияния на вызывающую программу.
- Изменение внутри функции аргумента, который является изменяемым объектом, может оказывать влияние на вызывающую программу.

Передача аргументов

Схема передачи аргументов:

- Неизменяемые объекты передаются «по значению».
- Изменяемые объекты передаются «по указателю».

Аргументы и разделяемые ссылки

```
>>> def f(a): # Имени a присваивается
                переданный объект
      a = 99 # Изменяется только локальная
               переменная
>>> b = 88
>>> f(b) # Первоначально имена a и b ссылаются
           на одно и то же число 88
>>>print(b)  # Переменная b не изменилась
88
```

Аргументы и разделяемые ссылки

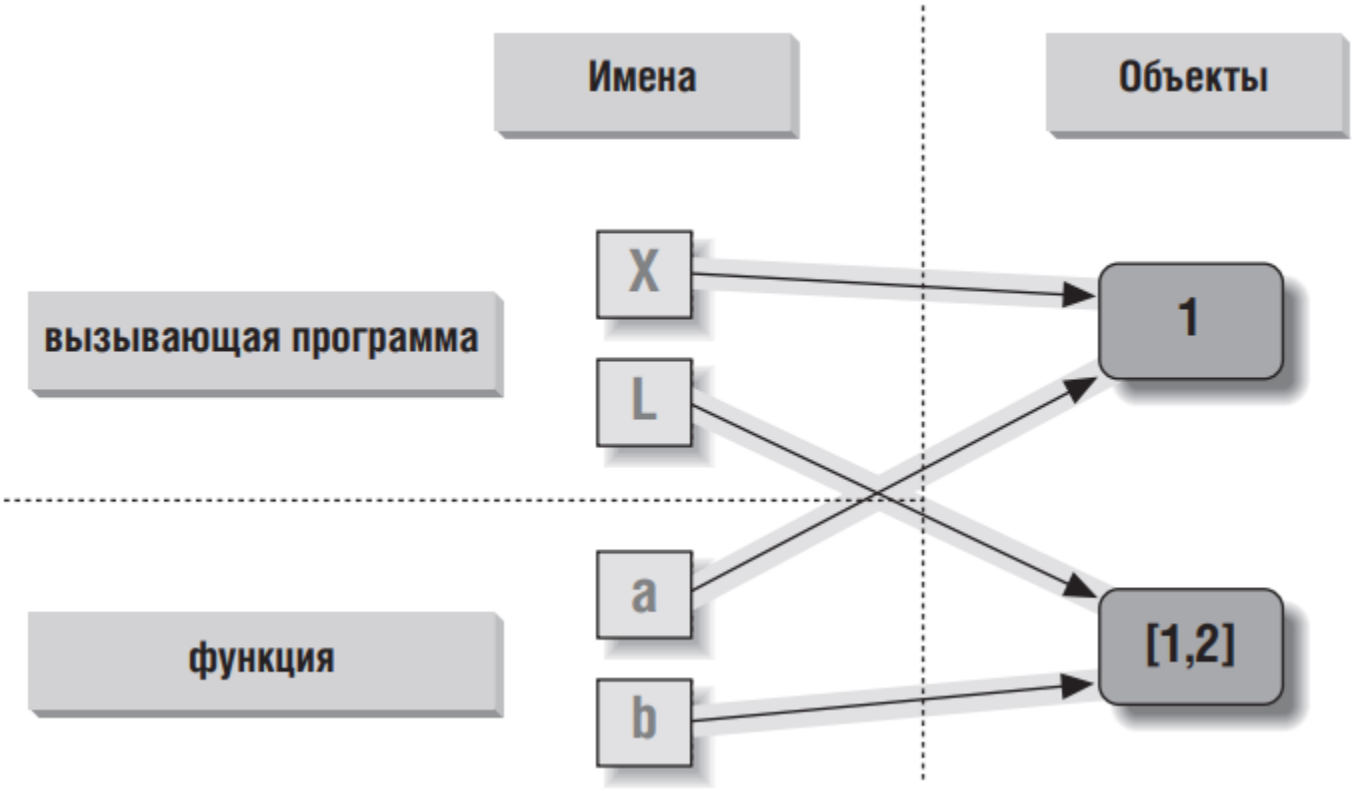
```
>>> def changer(a, b): # В аргументах передаются ссылки на объекты
        a = 2 # Изменяется только значение локального имени
        b[0] = ‘spam’ # Изменяется непосредственно разделяемый объект
>>> X = 1
>>> L = [1, 2]  # Вызывающая программа
>>> changer(X, L)  # Передаются изменяемый и неизменяемый объекты
>>>X, L  # Переменная X – не изменилась, L - изменилась
(1, [‘spam’, 2])
```

Аргументы и разделяемые ссылки

```
>>> X = 1
>>> a = X # Разделяют один и тот же объект
 >>> a = 2 # Изменяется только ‘a’, значение ‘X’ остается
                    равным 1
>>> print(X)
1

>>> L = [1, 2]
>>>b = L  # Разделяют один и тот же объект
>>>b[0] = ‘spam’  # Изменение в самом объекте: ‘L’ также
                                изменяется
 >>>print(L)
 [‘spam’, 2]
```

Ссылки: аргументы



Как избежать воздействий на изменяемые аргументы

```
L = [1, 2]
changer(X, L[:]) # Передается копия, поэтому
                 переменная 'L' не изменится

def changer(a, b):
        b = b[:] # Входной список копируется, что
                 исключает воздействие на
                 вызывающую программу
        a = 2
        b[0] = 'spam' # Изменится только копия
                      списка

L = [1, 2]
changer(X, tuple(L)) # Передается кортеж, поэтому
попытка изменения возбудит исключение
```

!!

Следует запомнить – функции могут оказывать воздействие на передаваемые им изменяемые объекты, такие как списки и словари. Это не всегда является проблемой и часто может приносить пользу.

Имитация выходных параметров

```
>>> def multiple(x, y):
        x = 2 # Изменяется только локальное имя
        y = [3, 4]
        return x, y # Новые значения возвращаются в виде кортежа
>>> X = 1
>>> L = [1, 2]
>>>X, L = multiple(X, L) # Результаты присваиваются именам в вызывающей программе
>>>X, L
(2, [3, 4])
```

Модули

import

Позволяет клиентам (импортерам) получать модуль целиком.

from

Позволяет клиентам получать определенные имена из модуля.

imp.reload

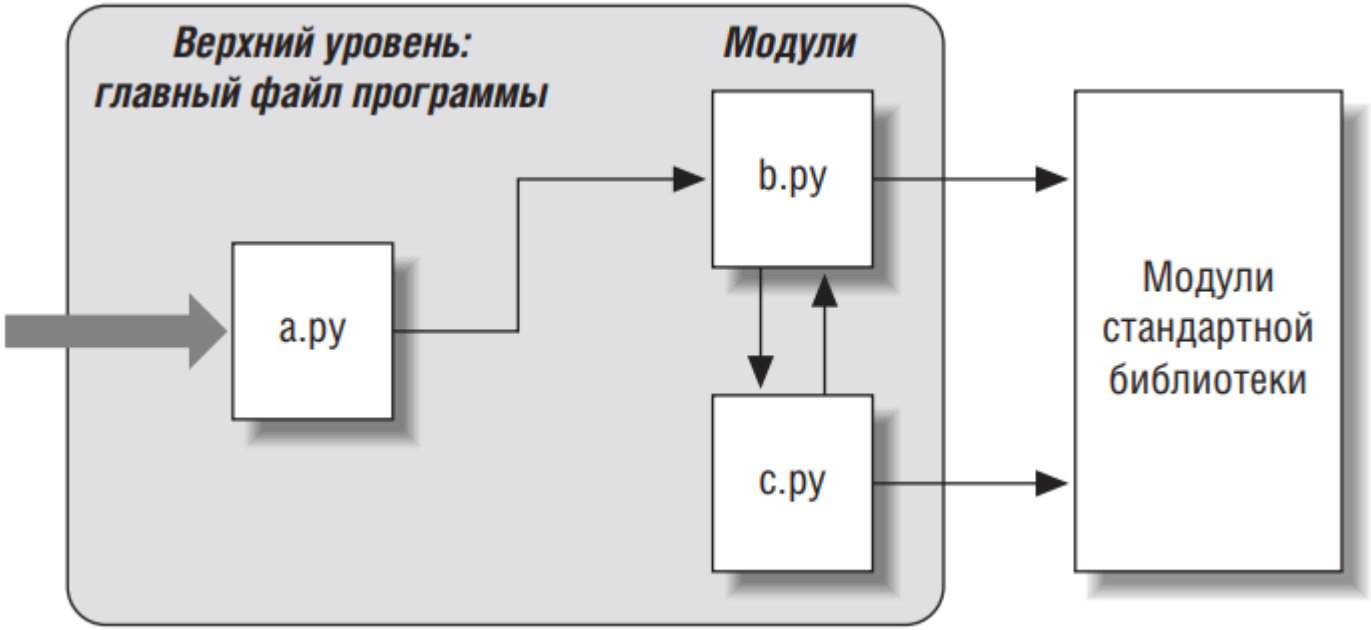
Обеспечивает возможность повторной загрузки модуля без остановки интерпретатора Python.

Зачем нужны модули

С точки зрения теории модули играют как минимум три роли:

- *Повторное использование программного кода*
- *Разделение системы пространств имен*
- *Реализация служб или данных для совместного пользования*

Архитектура программы на языке Python



b.py

```
def spam(text):
        print(text, 'spam')
```

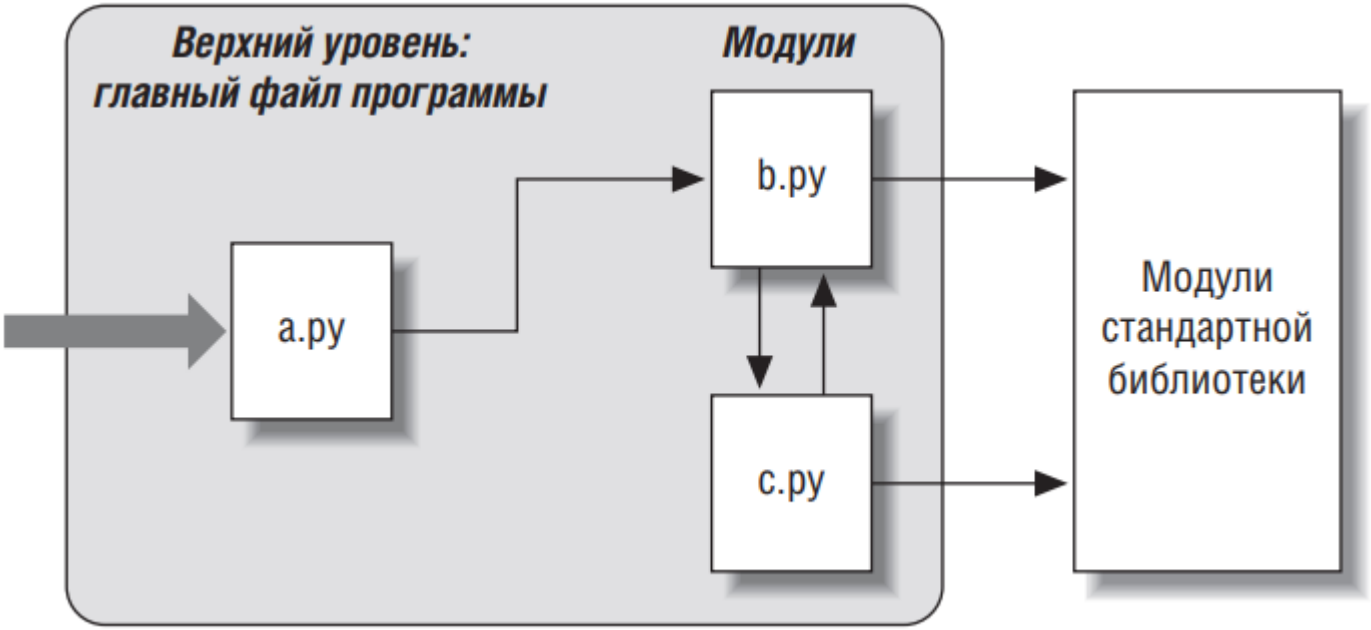
a.py

```
import b
b.spam('string')
  string spam
```

Архитектура программы на языке Python

Повсюду в сценариях на языке Python используется нотация **object.attribute** – большинство объектов обладают атрибутами, доступ к которым можно получить с помощью оператора «.»

Модули стандартной библиотеки



Создание модуля

текстовый файл с расширением «.py»

```
#module1.py
def printer(x):  # Атрибут модуля
        print(x)
```

if.py

import if

Использование модулей

Инструкции ***import*** или ***from***.

Обе инструкции отыскивают, компилируют и запускают программный код модуля, если он еще не был загружен.

Инструкция import

```
#module1.py
def printer(x):  # Атрибут модуля
        print(x)


>>> import module1 # Загрузить модуль целиком
>>> module1.printer('Hello world!') # Имя дополняется именем модуля Hello world!
```

Инструкция from

```
#module1.py
def printer(x):  # Атрибут модуля
        print(x)
```

```
>>>from module1 import printer  # Копировать одну переменную
>>>printer('Hello world!')  # Имя не требует дополнения
Hello world!
```

Инструкция from *

```
#module1.py
def printer(x):  # Атрибут модуля
        print(x)


>>> from module1 import * # Скопировать все
                                переменные
>>> printer('Hello world!')
Hello world!
```

Потенциальные проблемы инструкции from

- Инструкция ***from*** делает местоположение переменных менее явным и очевидным (имя ***name*** несет меньше информации, чем ***module.name***)
- Инструкция ***from*** способна повреждать пространства имен, по крайней мере, в принципе – если использовать ее для импортирования переменных, когда существуют одноименные переменные в имеющейся области видимости, то эти переменные просто будут перезаписаны.
- С другой стороны, инструкция **from** скрывает в себе более серьезные проблемы, когда используется в комбинации с функцией **reload**, так как импортированные имена могут ссылаться на предыдущие версии объектов.
- Кроме того, инструкция в форме **from module import *** действительно может повреждать пространства имен и затрудняет понимание имен, особенно когда она применяется более чем к одному файлу

Совет: отдавать предпочтение инструкции **import** для простых модулей, явно перечислять необходимые переменные в инструкциях **from** и не использовать форму **from *** для импорта более чем одного файла в модуле.

Когда необходимо использовать инструкцию import (1)

```
# M.py
def func():
        ...выполнить что-то одно...
# N.py
 def func():
        ...выполнить что-то другое...


# O.py
from M import func
from N import func # Перезапишет имя, импортированное из модуля M
func() # Будет вызвана N.func
```

Когда необходимо использовать инструкцию import (2)

```
# M.py
def func():
        ...выполнить что-то одно...
# N.py
 def func():
        ...выполнить что-то другое...

# O.py
import M,N  # Получить модуль целиком, а не отдельные имена
M.func()  # Теперь можно вызывать обе функции
N.func()  # Наличие имени модуля делает их уникальными
```