Программирование на Python

лекция 2

Владимир Юрьевич Полищук

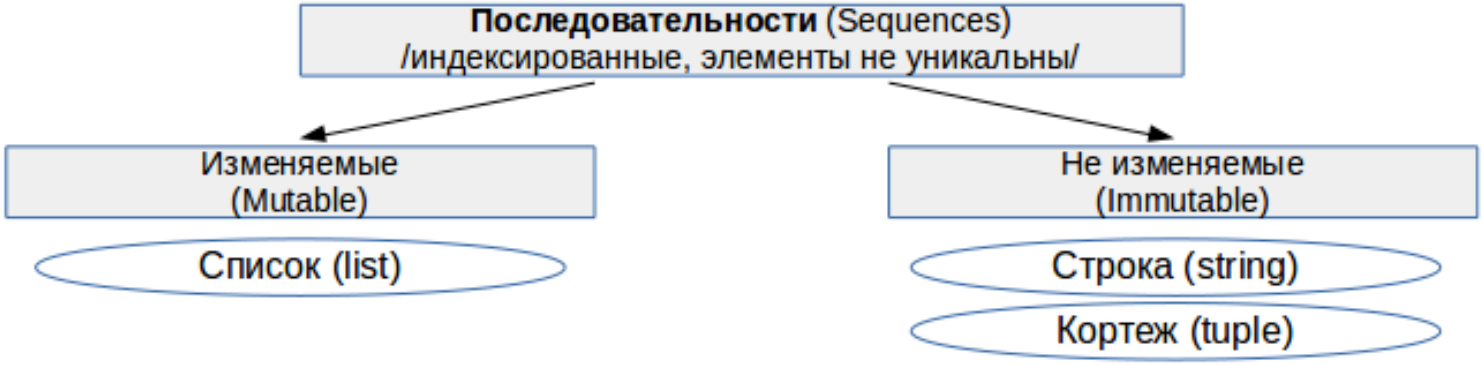
Инженерная школа
информационных технологий и
робототехники,
Томский политехнический университет

pvy3@tpu.ru

Содержание

- **Встроенные последовательности**
  - **Строки**
  - **Списки**
  - **Кортежи**
- **Динамическая типизация**

Встроенные последовательности



Операции над последовательностями

```
>>> S = ‘Word’  # 0 1 2 3
>>> len(S)     # Длина
4
>>> S[0]       # Первый элемент в W, счет
               начинается с позиции 0
‘W’
>>> S[1]       # Второй элемент слева
‘o’
```

Операции над последовательностями

```
>>> S = ‘Word’  # -4 -3 -2 -1
>>> S[-1]  # Последний элемент в конце
‘d’
>>> S[-2] # Второй элемент с конца
‘r’
>>> S[len(S)-1] # Отрицательная индексация,
                                более сложный способ
‘d’
```

Операции над последовательностями

Получение среза (slicing)

```
>>> S          #  Строка из 4 символов
'Word'         # 0 1 2 3
>>> S[1:3]     # Срез строки S начиная со смещения 1
                  и до 2 (не 3)
'or'
```

В общем виде синтаксис операции получения среза выглядит как **X[I:J]**, и означает: «**извлечь из X все, начиная со смещения I и до смещения J, но не включая его**».

Срезы

```
>>> S     # Сама строка S без изменений
'Word'
>>> S[1:] # Все, кроме первого элемента (1:len(S))
‘rod’
>>> S[0:3] # Все, кроме последнего элемента
'Wor'
>>> S[:3] # То же, что и S[0:3]
'Wor'
>>> S[:-1] # Еще раз все, кроме последнего элемента
'Wor'
>>> S[:] # Все содержимое S, как обычная копия (0:len(S))
'Word'
```

Операции над последовательностями

```
>>> S   # строка S без изменений
'Word'
>>> S + 'xyz'   # Конкатенация
'Wordxyz'

>>> S   # S остается без изменений
'Word'
>>> S * 3       # Повторение
'WordWordWord'
```

Неизменяемость

```
>>> S      # строка S без изменений
'Word'
>>> S[0] = 'z'        # Неизменяемые объекты нельзя
                      изменить
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    s[0]= 'Z'
TypeError: 'str' object does not support item assignment

>>> S = 'l' + S[1:]          # Но с помощью выражений мы
                             можем создавать новые объекты
>>> S
'lord'
```

Методы, специфичные для типа

```
>>> S.find('or')    # Поиск смещения подстроки
1

>>> S
'Word'

>>> S.replace('or', 'XYZ')  # Замена одной подстроки другой
'WXYZrd'

>>> S
'Word'
```

Методы, специфичные для типа

```
>>> line = ‘aaa, bbb, ccccc, dd’
>>> line.Split(‘,') # Разбивает строку по разделителю и создает список строк
['aaa', 'bbb', 'ccccc', 'dd']
>>> S = 'Word'
>>> S.upper()       #Преобразование символов в верхний регистр
'WORD'
>>> S.isalpha() # Проверка содержимого: isalpha, isdigit и так далее
True
>>> line = ‘aaa, bbb, ccccc, dd\n'
>>> line = line. rstrip() # Удаляет завершающие пробельные символы
>>> line
'aaa,bbb,ccccc,dd'
```

Методы, специфичные для типа

```
>>> '%s and %s' % ('word', 'WORD')
'word and WORD‘

>>> '{0} and {1}'.Format(‘word', 'WORD')
'word and WORD'
```

!!

Строковые методы могут применяться только к строкам и ни к каким другим объектам.

Инструментальные средства языка **Python** делятся на несколько уровней:

- ***универсальные*** операции, которые могут применяться к нескольким типам, реализованы в виде встроенных функций и выражений (например, len(X), X[0]),
- ***специфичные*** для определенного типа, реализованы в виде методов (например, Str.upper()).

Получение помощи

>>> dir(s)

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

Получение помощи

```
>>> help(s.isalpha)
Help on built-in function isalpha:

isalpha() method of builtins.str instance
    Return True if the string is an alphabetic string, False otherwise.

    A string is alphabetic if all characters in the string are alphabetic and there
    is at least one character in the string.
```

Списки

Списки – это упорядоченные по местоположению коллекции объектов произвольных типов, размер которых не ограничен.

Списки являются изменяемыми.

Операции над последовательностями

```
>>> L = [123, 'word', 1.23] # Список из трех объектов разных типов
>>> len(L)          # Число элементов в списке
3
>>> L[0] # Доступ к элементу списка по его индексу
123
>>> L[:-1]          # Операция получения среза возвращает новый
                    список
[123, 'word']
>>> L + [4, 5, 6] # Операция конкатенации также возвращает новый
                  список
[123, 'word', 1.23, 4, 5, 6]
>>> L     # Наши действия не привели к изменению оригинального
          списка
[123, 'word', 1.23]
```

Методы, специфичные для типа

```
>>> L.append('N1') # Увеличение: в конец списка добавляется новый объект
>>> L
[123, 'word', 1.23, 'N1']
>>> L.pop(2)  # Уменьшение: удаляется элемент из середины списка
[123, 'word', 'N1']
>>> del L[2]    # Инструкция также удалит элемент списка
[123, 'word', 'N1']
```

Методы, специфичные для типа

```
>>> M = ['bb', 'aa’, 'cc']
>>> M.sort()
>>> M
['aa', ‘bb', 'cc']
>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

Вложенные списки

```
>>> M = [[1, 2, 3],          # Матрица 3 x 3 в виде вложенных списков
         [4, 5, 6],           # Выражение в квадратных скобках может
         [7, 8, 9]]           # занимать несколько строк
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> M[1] # Получить строку 2
[4, 5, 6]
>>> M[1][2] # Получить строку 2, а затем элемент 3 в этой строке
6
```

Генераторы списков

```
>>> col2 = [row[1] for row in M] # Выбирает элементы второго столбца
>>> col2
 [2, 5, 8]
>>> M # Матрица не изменилась
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]

>>> [row[1] + 1 for row in M] # Добавить 1 к каждому элементу в столбце 2
[3, 6, 9]
>>> [row[1] for row in M if row[1] % 2 == 0] # отфильтровать  нечетные значения
[2, 8]
```

Генераторы списков

```
>>> diag = [M[i][i] for i in [0, 1, 2]] # Выборка элементов диагонали матрицы
>>> diag
 [1, 5, 9]
>>> doubles = [c * 2 for c in 'word'] # Дублирование символов в строке
>>> doubles
['ww', 'oo', 'rr', 'dd']
```

Генераторы списков

```
>>> G = (sum(row) for row in M) # Генератор, возвращающий суммы элементов строк
>>> next(G)
6
>>> next(G)    # Вызов в соответствии с протоколом итераций
15

>>> list(map(sum, M)) # Отобразить sum на элементы в M
[6, 15, 24]
```

Генераторы списков

```
>>> {sum(row) for row in M}      # Создаст множество сумм строк
{24, 6, 15}
>>> {i: sum(M[i]) for i in range(3)} # Таблица пар ключ/значение сумм строк
{0: 6, 1: 15, 2: 24}
```

Генераторы списков

```
>>> [ord(i) for i in 'word'] # Список кодов символов
[119, 111, 114, 100]
>>> {ord(i) for i in 'wooord'} # Множества ликвидируют дубликаты
{100, 114, 111, 119}
>>> {i:ord(i) for i in 'wooord'} # Ключи словарей являются уникальными
{'w': 119, 'o': 111, 'r': 114, 'd': 100}
```

!!

() - **кортеж**

Кортеж - это последовательность элементов, которые нельзя изменить (**неизменяемые**).

[] - **список**

Список - это последовательность элементов, которые можно изменить (**изменяемые**).

{} - словарь или набор

Словарь - это список пар ключ-значение с уникальными ключами (**изменяемые**).

Словари

Словари *(отображения) – это коллекции объектов, но доступ к ним осуществляется не по определенным смещениям от начала коллекции, а по ключам.*

Словари – единственный тип отображения в наборе базовых объектов Python – также относятся к классу изменяемых объектов.

Операции над отображениями

Последовательность пар **{ключ: значение}**

Удобно использовать когда нужно **описать свойства чего-либо**.

>>> D = {‘food’: ‘apple’, ‘quantity’: 4, ‘color’: ‘red’}

Операции над отображениями

```
>>> D[‘food'] # Получить значение, связанное с ключом 'food'
‘apple’

>>> D['quantity'] += 1 # Прибавить 1 к значению ключа 'quantity‘

>>> D
{'food': ‘apple', 'color': ‘red', 'quantity': 5}
```

Операции над отображениями

```
>>> D = {}
>>> D['name'] = 'Вася' # В результате присваивания создается ключ
>>> D['occupation’] = 'студент'
>>> D['age' ] = 20
>>> D
{'age': 20, 'occupation': 'студент', 'name': 'Вася'}
>>> print(D[' name' ])
Вася
```

О вложенности

```
>>> rec = {‘name’: {‘first’: ‘Вася’, ‘last’: ‘Иванов’},
‘occupation’: [‘студент’, ‘аспирант’], ‘age’: 20}
```

О вложенности

```
>>> rec['name'] # 'Name' - это вложенный словарь
{'last' : 'Иванов', 'first' : ‘Вася’}
>>> rec[' name' ][' last' ] # Обращение к элементу вложенного словаря
'Иванов'
>>> rec['occupation']     # ‘occupation’ - это вложенный список
[‘студент’, ‘аспирант’]
>>> rec[‘occupation’][-1] # Обращение к элементу вложенного списка
‘аспирант’
>>> rec[‘occupation’].append('магистр') # Расширение списка деятельностей Васи
>>> rec
{'age': 20, ' occupation': [‘студент’, ‘аспирант’, 'магистр'], 'name': {'last': ‘Иванов’, 'first' : ‘Вася’}}
```

!!

```
>>> rec = 0 # Теперь память, занятая объектом, будет освобождена
```

Сортировка по ключам: циклы for

```
>>> D = {‘a’: 1, ‘b’: 2, ‘c’: 3}
>>> D
{‘a’: 1, ‘c’: 3, ‘b’: 2}
```

Сортировка по ключам: циклы for

```
>>> newL = list(D. keys()) # Неупорядоченный список ключей
>>> newL
[' a', ' c', ' b']
>>> newL.sort() # Сортировка списка ключей
>>> newL
['a', 'b’, 'c']
>>> for key in newL: # Обход отсортированного списка ключей
print(key, '=>', D[key])
a => 1
b => 2
c => 3
```

Сортировка по ключам: циклы for

```
>>> D
{‘a’: 1, ‘c’: 3, ‘b’: 2}
>>> for key in sorted(D): print(key, ‘=>’, D[key])
a => 1
b => 2
c => 3
```

Циклы for и while

```
>>> for c in ‘word’: print(c.upper())
W
O
R
D

>>> x = 3
>>> while x > 0:
        print(‘word!’ * x)
        x -= 1
word! word! word!
word! word!
word!
```

Итерации

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
>>> squares
[1, 4, 9, 16, 25]

>>> squares = []
>>> for x in [1, 2, 3, 4, 5]: # Эти же операции выполняет и генератор списков,
        squares.append(x ** 2) # следуя протоколу итераций
 >>> squares
 [1, 4, 9, 16, 25]
```

Кортежи

Объект-кортеж (tuple – произносится как «тъюпл» или «тъюпел», в зависимости от того, у кого вы спрашиваете) в общих чертах напоминает список, который невозможно изменить – кортежи являются последовательностями, как списки, но они являются неизменяемыми, как строки.

```
>>> T = (1, 2, 3, 4)  # Кортеж из 4 элементов
>>> len(T)
4
>>> T + (5, 6) # Конкатенация
(1, 2, 3, 4, 5, 6)
>>> T[0]   # Извлечение элемента
1
```

Кортежи

```
>>> T.index(4) # Методы кортежей: значение 4 находится в позиции 3
3
>>> T.count(4) # Значение 4 присутствует в единственном экземпляре
1
>>> T[0] = 2 # Кортежи являются неизменяемыми
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    T[0]=2
TypeError: 'tuple' object does not support item assignment
```

Кортежи

```
>>> t=('word', 3.0, [11,22,33],{3,4})
>>> t[1]
3.0
>>> t[0]
'word'
>>> t.append(4)
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    t.append(4)
AttributeError: 'tuple' object has no attribute
'append'
```

!!

Зачем нужны кортежи?

На практике кортежи используются не так часто, как списки, и главное их достоинство – неизменяемость.

Кортежи обеспечивают своего рода ограничение целостности, что может оказаться полезным в некоторых программах.

Обобщение свойств встроенных коллекций

Тип коллекции	Изменяемость	Индексированность	Уникальность	Как создаём
Список (list)	+	+	-	*[]* *list()*
Кортеж (tuple)	-	+	-	*() ,* *tuple()*
Строка (string)	-	+	-	*''* *""*
Словарь (dict)	+ элементы - ключи + значения	-	+ элементы + ключи - значения	*{}* *{key: value,}* *dict()*

Общие подходы к работе с любой коллекцией

```
>>>my_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>>my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}


>>>print(my_list)
['a', 'b', 'c', 'd', 'e', 'f']
>>>print(my_dict)
{'a': 1, 'c': 3, 'e': 5, 'f': 6, 'b': 2, 'd': 4} # Не забываем, что порядок элементов в неиндексированных коллекциях не сохраняется.

print(len(my_list)) # 6
print(len(my_dict)) # 6 - для словаря пара ключ-значение считаются одним элементом.
print(len('ab c')) # 4 - для строки элементом является 1 символ
```

Общие подходы к работе с любой коллекцией

```
>>>my_list = ['a', 'b', 'c', 'd', 'e', 'f']

>>>print('a' in my_list)
True
>>> print('q' in my_list)
False
>>> print('a' not in my_list)
False
>>> print('q' not in my_list)
True
```

Общие подходы к работе с любой коллекцией

```
>>>my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}

>>>print('a' in my_dict) # без указания метода поиск по ключам
 True
>>> print('a' in my_dict.keys()) # аналогично примеру выше
True
>>> print('a' in my_dict.values()) # так как 'a' — ключ, не значение
 False
>>> print(1 in my_dict.values())
True
```

Общие подходы к работе с любой коллекцией

```
>>>print(('a',1) in my_dict.items())
True
>>>print(('a',2) in my_dict.items())
False


>>>print('ab' in 'abc')
```

Обход всех элементов коллекции в цикле for

```
>>> for elm in my_list:
        print(elm)


>>> for elm in my_dict:
        # При таком обходе словаря, перебираются только ключи
        # равносильно for elm in my_dict.keys()
      print(elm)

>>> for elm in my_dict.values():
        # При желании можно пройти только по значениям
      print(elm)


>>> for key, value in my_dict.items():
          # Проход по .items() возвращает кортеж (ключ, значение),
          # который присваивается кортежу переменных key, value
       print(key, value)
```

Функции min(), max(), sum()

```
>>>my_list = ['a', 'b', 'c', 'd', 'e', 'f']
>>>my_dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5, 'f': 6}


>>>print(min(my_list))
a
>>>print(sum(my_dict.values()))
21
```

Динамическая типизация
Отсутствие инструкций объявления

Типы данных в языке Python определяются автоматически во время выполнения, это означает, что вам не требуется заранее объявлять переменные.

Переменные, объекты и ссылки

a = 3

- *Создание переменной*
- *Типы переменных*
- *Использование переменной*

В итоге складывается следующая картина:

- переменные создаются при выполнении операции присваивания,
- могут ссылаться на объекты любых типов
- переменным должны быть присвоены некоторые значения, прежде чем к ним можно будет обратиться.

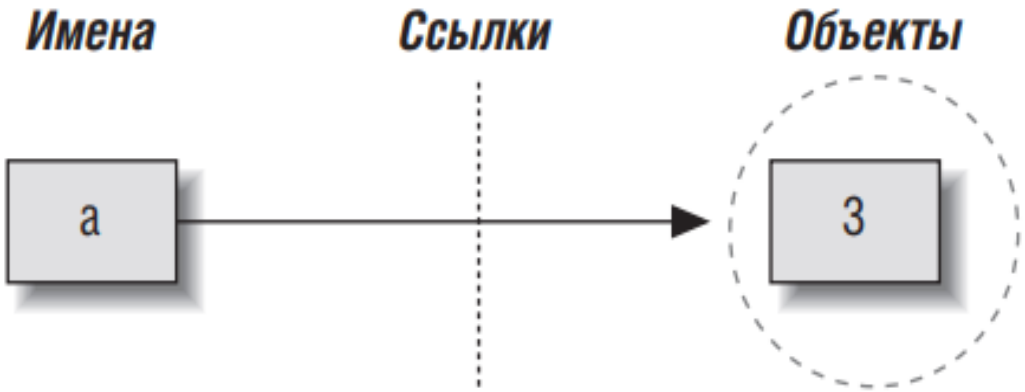
Это означает, что от вас не требуется заранее объявлять переменные в сценарии, но вы должны **инициализировать их перед использованием** – счетчик, например, должен быть инициализирован нулевым значением, прежде чем его можно будет наращивать.

Модель динамической типизации

>>> a = 3

1. Создается объект, представляющий число 3.
2. Создается переменная **a**, если она еще отсутствует.
3. В переменную **a** записывается ссылка на вновь созданный объект, представляющий число 3.

Модель динамической типизации



Переменные – это записи в системной таблице, где предусмотрено место для хранения ссылок на объекты.

Объекты – это области памяти с объемом, достаточным для представления значений этих объектов.

Ссылки – это автоматически разыменовываемые указатели на объекты.

!!

Каждый объект имеет два стандартных поля:

- описатель типа, используемый для хранения информации о типе объекта;
- счетчик ссылок, используемый для определения момента, когда память, занимаемая объектом, может быть освобождена.

Информация о типе хранится в объекте, но не в переменной

```
>>> a = 3          # Это целое число
>>> a = 'word'     # теперь это строка
>>> a = 42.2       # теперь это вещественное
                   число
```

- Переменные не имеют типов;
- Тип – это свойство объекта;
- В действительности мы не изменяем типы переменных – мы просто записываем в переменные ссылки на объекты других типов.

На самом деле, все, что можно сказать о *переменных* в языке Python, – это то, что *они ссылаются на конкретные объекты в конкретные моменты времени*.

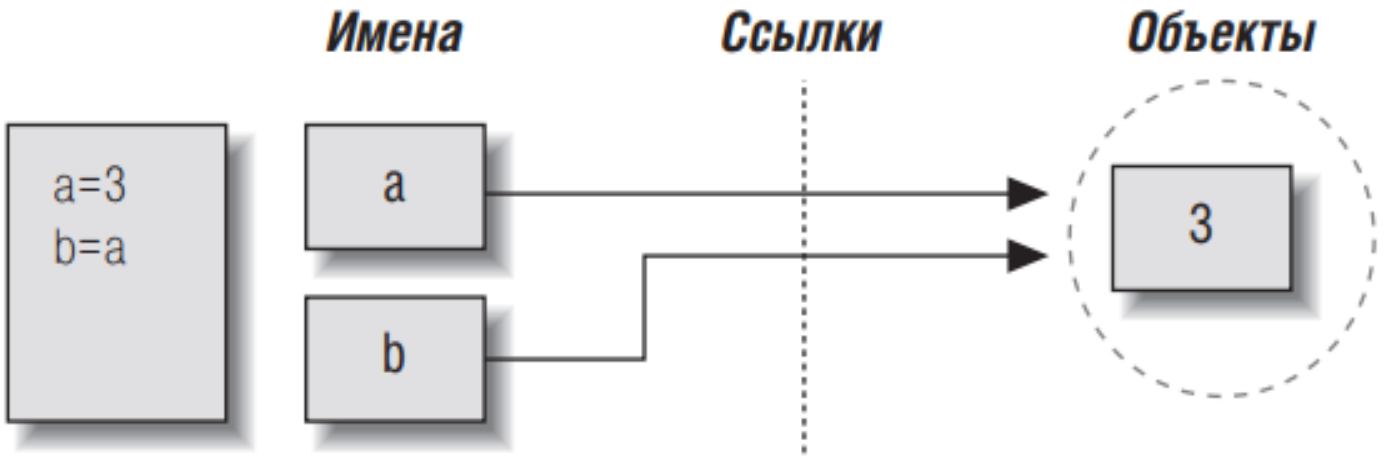
Объекты уничтожаются автоматически

Когда имя ассоциируется с новым объектом, интерпретатор Python освобождает память, занимаемую предыдущим объектом (если на него не ссылается какое-либо другое имя или объект). Такое автоматическое освобождение памяти, занимаемой объектами, называется сборкой мусора (garbage collection).

```
>>> x = 42
>>> x = ‘object’ # Освобождается объект 42 (если нет других ссылок)
>>> x = 3.1415 # Теперь освобождается объект ‘object’
>>> x = [1,2,3] # Теперь освобождается объект 3.1415
```

Разделяемые ссылки

```
>>> a = 3
>>> b = a
```

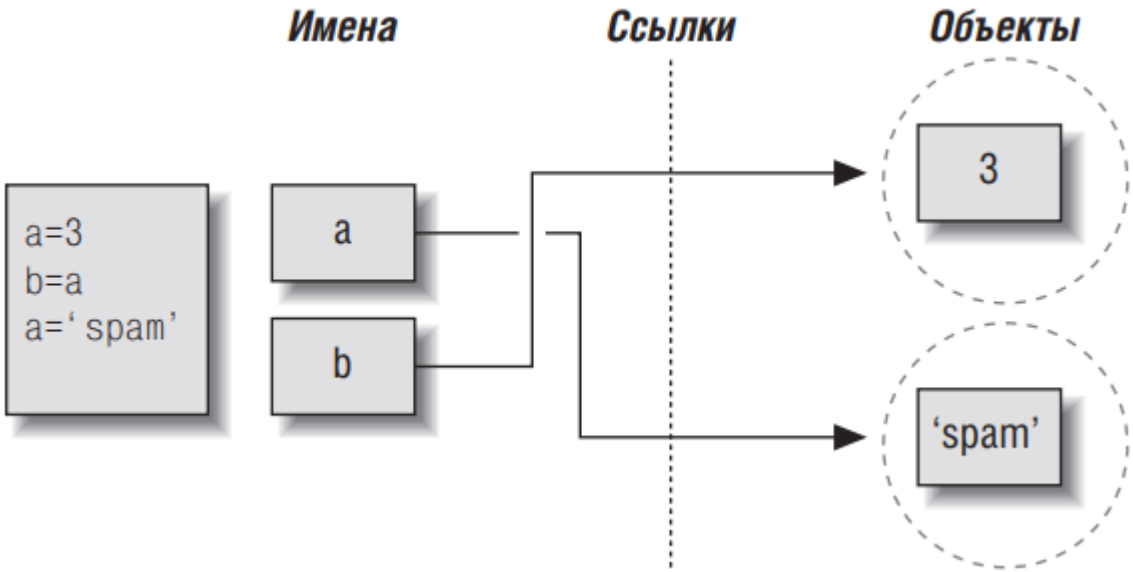


Разделяемые ссылки

>>> a = 3

>>> b = a

>>> a = ‘spam’



Разделяемые ссылки

```
>>> a = 3
>>> b = a
>>> a = a + 2
```

Это не приводит к изменению переменной **b**. В действительности нет никакого способа перезаписать значение объекта **3**.

Запись нового значения в переменную не приводит к изменению первоначального объекта, но приводит к тому, что переменная начинает ссылаться на совершенно другой объект

Разделяемые ссылки и изменяемые объекты

```
>>> L1 = [2, 3, 4] # Изменяемый объект
>>> L2 = L1 # Создание второй ссылки на тот же самый объект
>>> L1[0]
2
>>> L1 = 24

>>> L1[0] = 24   # Изменение объекта
>>> L1           # Переменная L1 изменилась
[24, 3, 4]
>>> L2           # Но также изменилась и переменная L2
[24, 3, 4]
```

Разделяемые ссылки и изменяемые объекты

```
>>> L1= [2, 3, 4]
>>> L2 = L1[:] # Создается копия списка L1
>>> L1[0] = 24
>>> L1
[24, 3, 4]
>>> L2 # L2 не изменилась
[2, 3, 4]
```

Здесь изменения в **L1** никак не отражаются на **L2**, потому что **L2** ссылается на копию объекта, на который ссылается переменная **L1**. То есть эти *переменные указывают на различные области памяти*.

Разделяемые ссылки и изменяемые объекты

```
import copy
X = copy.copy(Y)      # Создание "поверхностной" копии
                      любого объекта Y
X = copy.deepcopy(Y) # Создание полной копии:
                              копируются все вложенные
```

Разделяемые ссылки и равенство

```
>>> x = 42
>>> x = 'object' # Объект 42 теперь уничтожен?

>>> L = [1, 2, 3]
>>> M = L     # M и L - ссылки на один и тот же
              объект
>>> L == M    # Одно и то же значение
True
>>> L is M    # ссылаются ли на один и тот же
              объект
True
```

Разделяемые ссылки и равенство

```
>>> L = [1, 2, 3]
>>> M = [1, 2, 3] # M и L ссылаются на разные объекты
>>> L == M          # Одно и то же значение
True
>>> L is M          # Но разные объекты
False

>>> X = 42
>>> Y = 42 # Должно получиться два разных объекта
>>> X == Y
True
>>> X is Y # Тот же самый объект: кэширование в действии!
True
```

Разделяемые ссылки и равенство

```
>>> import sys
>>> sys.getrefcount(0)      # 976 указателей на этот участок памяти
976
```

!!

Мы подробно рассмотрели **модель динамической типизации** в языке Python, то есть способ, который используется интерпретатором для автоматического хранения информации о типах объектов и избавляет нас от необходимости вставлять код объявлений в наши сценарии.

Попутно мы узнали, как реализована связь между переменными и объектами. Кроме того, мы исследовали понятие сборки мусора, узнали, как разделяемые ссылки на объекты могут оказывать воздействие на несколько переменных и как ссылки влияют на понятие равенства в языке Python.

Поскольку в языке Python имеется всего одна модель присваивания, и она используется повсюду в этом языке, очень важно, чтобы вы разобрались в ней, прежде чем двигаться дальше