Python — очень доступный язык. Чаще всего он работает именно так, как вы ожидаете. Но порой его поведение может вас удивлять — особенно, если раньше вы пользовались другими языками.

Внутренние механизмы Python зачастую довольно просты, но эффект от их комбинации может быть весьма неожиданным. Тем не менее, разобравшись в механизмах языка, вы сможете понять и причины того или иного поведения программы.

Сегодня мы поговорим об именах, значениях и изменяемости (мутабельности).

Имена ссылаются на значения

Как и во многих других языках, присваивание в Python — это связывание символического имени в левой части выражения со значением в правой части. Если имени присвоено какое-нибудь значение, мы говорим, что оно ссылается на это значение или является ссылкой на значение.

```
x = 23
```

Теперь имя `x` ссылается на значение 23. В дальнейшем при использовании имени `x` мы будем получать значение 23.

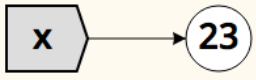
```
print(x) # выводит 23
```

Также можно сказать, что имя `x` привязано к значению 23.

То, как именно имя ссылается на значение, на самом деле не слишком важно. Если вы имеете опыт работы с языком C, можете считать, что это указатель. Но если это вам ни о чем не говорит, — не заморачивайтесь.

Схематические изображения в этой статье помогут вам разобраться в происходящем в коде. Серый ярлычок — это имя, а стрелочка указывает на его значение. На иллюстрации ниже вы видите, что имя x ссылается на целое число 23.

```
x = 23
print(x)        # 23
```



Все схемы — из оригинальной статьи, автор — nedbat

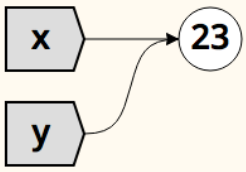
На одно значение может ссылаться несколько имен

Нет никаких правил, запрещающих значению иметь больше одного имени. При помощи присваивания можно назначить это значение и второму, и третьему (и т. д.) имени.

```
x = 23

y = x
```

Теперь и `x`, и `y` ссылаются на одно значение.

```
x = 23
y = x
```



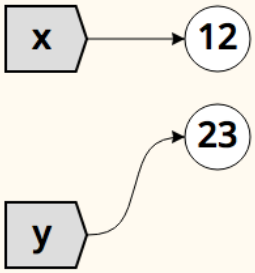
При этом нельзя сказать, что какое-то из имен «настоящее». У них одинаковый статус: они ссылаются на одно и то же значение совершенно одинаковым образом.

Сколько бы имен ни ссылалось на одно значение, каждому из них всегда можно присвоить другое значение независимо от остальных

Если два имени ссылаются на одно значение, это не устанавливает никакой магической связи между самими именами. Присвоив одному из имен новое значение, вы никоим образом не затронете другое.

```
x = 23

y = x

x = 12
```

```
x = 23
y = x
x = 12
```



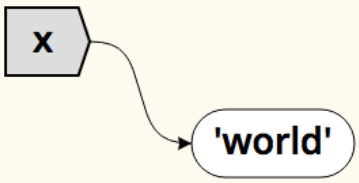
Утверждая, что `y = x`, мы не имеем в виду, что они всегда будут тождественны. Присвойте `x` новое значение — и `y` останется в одиночестве. Вы только представьте, какой хаос был бы в противном случае!

Важно отметить, что Python не позволяет сделать одно имя ссылкой на другое, у него просто нет подобного механизма. Таким образом, вы не сможете сделать `y` постоянным псевдонимом для `x`, не теряющим свой статус даже при смене значения `x`.

Значения «живут», пока на них ведет хоть одна ссылка

Python отслеживает, сколько ссылок ведет на каждое значение, и автоматически очищает значения, на которые вообще не ведут никакие ссылки. Это называется «сборкой мусора» и означает, что вам не нужно избавляться от ненужных значений вручную. Когда необходимость в значениях отпадет, они исчезнут сами.

```
x = "hello"
x = "world"
```



Как именно Python все это отслеживает, относится к деталям реализации. Но, возможно, вы слыхали термин «подсчет ссылок». Эта техника является важной частью механизма сборки мусора.

Присваивание не копирует данные

Важный факт о присваивании: присваивание никогда не копирует данные.

Если у значения два имени, можно ошибочно предположить, что значений тоже два:

```
x = 23

y = x

# "Теперь у меня два значения: x и y!"

# НЕТ: у вас два имени, а значение только одно.
```

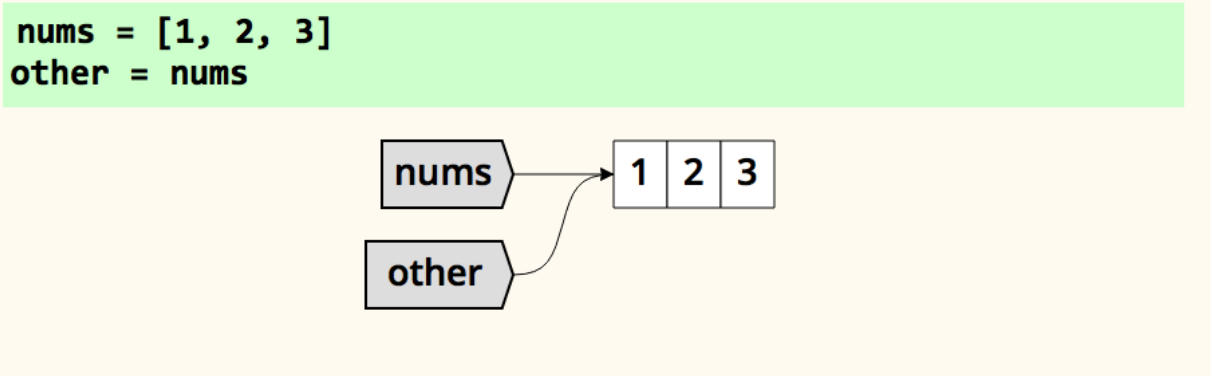
Присваивание значения никогда не копирует данные, новое значение при этом никогда не создается. Выполняя присваивание, вы просто делаете так, чтобы имя в левой части ссылалось на значение в правой. Если обратиться к нашему примеру, то у нас есть только значение 23, а `x` и `y` — оба! — ссылаются на него.

Все становится куда интереснее, когда речь заходит о более сложных значениях, например, о списках.

```
nums = [1, 2, 3]
```

Если мы теперь присвоим `nums` другое имя, у нас будут два имени, ссылающиеся на один список:

```
nums = [1, 2, 3]

other = nums
```

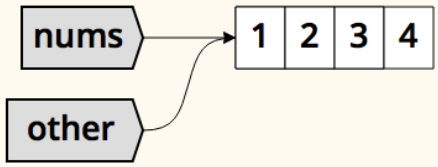


Помните: присваивание никогда не создает новых значений и не копирует данных. И операция присваивания в нашем примере не превращает один список в два.

На этом этапе у нас есть один список, на который ссылаются два имени. И здесь нас может ожидать сюрприз.

Изменения в значении видны всем именам

```
nums = [1, 2, 3]
other = nums
nums.append(4)
print(other)        # [1, 2, 3, 4] !!!
```



В этом примере мы заново создали наш список и назначили его в качестве значения двум именам — `nums` и `other`. После этого мы добавили еще одно значение в список `nums`. При выводе на экран списка `other` оказалось, что четверка добавилась и в этот список. Многих людей это удивляет.

Но на самом деле удивляться нечему. Оба имени указывали на один список, поэтому изменения, внесенные в список под одним именем, отразились и при выводе значения второго. Присваивание значению [1, 2, 3] второго имени — `other` — не создало новый список. И перед операцией добавления значения копия списка тоже не создалась. Список только один, и если вы вносите в него изменения, обращаясь по одному из имен, эти изменения будут отражаться при выводе всех остальных имен.

Изменяемые значения могут иметь дополнительные имена

Это достаточно важный эффект. При этом он часто оказывается неожиданностью для людей, так что имеет смысл повторить еще раз. Если изменяемый объект имеет больше одного имени, а значение изменяется, эти изменения увидят все имена.

В маленьком примере, который мы рассматривали, легко увидеть, что у нас два имени для одного списка. Но как мы увидим на примерах вызова функций, два имени могут появляться довольно далеко друг от друга. А изменение значения где-то далеко от внесения правок может оказаться сюрпризом.

Здесь мы впервые встречаемся с термином «*изменяемый*» (или «мутабельный», от англ. mutable). *Мутабельность* означает, что значение может изменяться. В нашем

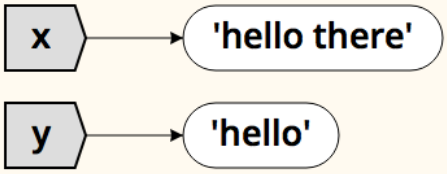
примере имя `nums` постоянно ссылается на один и тот же объект — во всех четырех строчках кода. Но значение, содержащееся в этом объекте, меняется.

Неизменяемые значения

Не все значения в Python изменяемы. Числа, строки и кортежи — все они относятся к неизменяемым. Нет никакой операции, которая позволяла бы менять их. Все, что вы можете сделать, это создать новый объект из старых.

Неизменяемые типы:
числа (int и float), строки (string), кортежи (tuple)

```
x = "hello"
y = x
x = x + " there"
```



В нашем примере `x` ссылается на строку “hello”. Затем `y` тоже ссылается на эту строку. В последней строке оператор `+` не расширяет существующую строку, а создает совершенно новую путем конкатенации “hello” и ” there”. После этого `x` ссылается уже на новую строку.

Старая строка остается неизменной. Со строками так будет всегда, ведь они неизменяемые. Нет никаких методов для модификации строк: все, что есть, возвращают новые строки.

Проблема дополнительных имен здесь вообще не стоит, ведь значение вы все равно не сможете изменить.

Что мы подразумеваем под словом «изменение»

При обсуждении вопроса изменяемости важно остановиться на самом слове «изменение».

Неформально мы говорим, что добавление 1 к `x` «изменяет» `x`:

```
x = x + 1
```

Мы также говорим, что добавление новых элементов в `num` «изменяет» `num`:

```
num.append(7) # изменяет num
```

Но это две совершенно разные операции. Первая — *повторное связывание* `x`. `x+1` образует совершенно новый объект, а затем этому объекту присваивается имя `x`.

Добавление элементов в `num` — это *мутация* `num`. Имя `num` указывает на все тот же объект, но сам объект был модифицирован, его значение обновилось.

Было бы слишком странно постоянно оперировать терминами «повторное связывание» и «мутация». Но когда мы читаем код и пытаемся в нем разобраться, эти слова могут помочь разграничить разные виды изменений.

Конечно, вы также можете применить повторное связывание для имени, ссылающегося на список. Это еще один способ сделать так, чтобы `nums` стал списком с 7 в конце:

```
nums = nums + [7]
```

Как и в случае с числами, здесь оператор `+` создает совершенно новый список, а затем имя `nums` присваивается уже новому списку.

А вот изменить число путем мутации мы не можем. Числа неизменяемы. В английском этот термин — *immutable* — буквально означает невозможность мутации.

Изменяемые и неизменяемые значения присваиваются одинаково

Одно из распространенных заблуждений относительно Python состоит в том, что присваивание работает по-разному для изменяемых и неизменяемых значений. Это неправда. Присваивание — очень простая операция, в ее результате имя с левой стороны выражения начинает ссылаться на значение с правой стороны.

Когда люди говорят, что присваивание работает по-разному, они неправильно диагностируют проблему дополнительных имен у изменяемых значений. Они видят в двух отрывках кода два разных эффекта и знают, что в одном случае значение было изменяемым, а во втором — нет. В результате люди ошибочно заключают, что сам шаг присваивания чем-то отличался.

Но внутренние механизмы Python на самом деле намного проще, чем многие думают. Операция присваивания всегда предполагает одно и то же действие, независимо от значения, стоящего в правой части выражения.

Присваивание может выглядеть по-разному

Помимо простого знака равенства в Python есть и другие виды присваивания. Например, и для чисел, и для списков можно использовать `+=`.

По сути, следующие две строки кода одинаковы:

```
x += y

x = x + y
```

Работа `+=` в Python реализована через значение x. Эти две строки эквивалентны:

```
x += y

x = x.__iadd__(y)
```

Значение `+=` зависит от типа `x`, потому что это значение определяет реализацию `__iadd__`, который будет использоваться.

С числами `+=` будет работать точно так, как вы и ожидаете. Но со списками вас ждет очередной сюрприз. Со списками `nums = nums + more` свяжет имя `nums` с новым списком, сформированным путем конкатенации `nums` и `more`. А `nums += more` модифицирует сам список `nums` (мутация).

Причина этого в том, что список реализует `__iadd__` следующим образом:

```
class List:

def __iadd__(self, other):

self.extend(other)

return self
```

Когда вы выполняете `nums += more`, вы получаете то же, что и при следующем присваивании:

```
nums = nums.__iadd__(more)
```

которое, в силу реализации `__iadd__`, работает так:

```
nums.extend(more)

nums = nums
```

Так что здесь есть операция повторного связывания, но сначала происходит операция мутации.

Мораль: нужно разбираться в поведении примитивов, которые используете!

Ссылки могут быть чем-то большим, чем просто именами

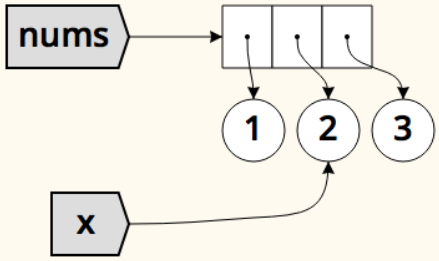
Все примеры, которые мы пока рассматривали, в качестве ссылок на значения использовали имена. Но ссылками могут быть и другие вещи.

В Python есть сложные структуры данных, каждая из которых хранит ссылки на значения: списки элементов, ключи словарей, атрибуты объектов и т. д. Все перечисленное может использоваться с левой стороны выражения присваивания, и все, о чем мы уже говорили, останется в силе.

Что бы ни оказалось с левой стороны выражения присваивания, оно станет ссылкой. Здесь и далее, если мы говорим «имя», вы вполне можете заменять его на «ссылку».

На схемах списков, которые мы рассматривали до сих пор, вы видели числа, хранящиеся в ячейках. Но на самом деле каждый элемент списка — это ссылка. Так что список можно изобразить следующим образом (стрелочки, исходящие из ячеек, указывают на значения):

```
nums = [1, 2, 3]
x = nums[1]
```



Но так можно быстро запутаться, поэтому мы используем более простую схему, где числа изображены в самих ячейках.

Ссылками могут быть многие сущности…

Вот несколько примеров присваиваний. Все, что находится в левой части выражений, — ссылки.

```
my_obj.attr = 23

my_dict[key] = 24
```

```
my_list[index] = 25

my_obj.attr[key][index].attr = "etc, etc"
```

И так далее. Многие структуры данных Python содержат значения, и каждое из этих значений — ссылка. Все правила относительно имен касаются всех этих ссылок. Например, когда сборщик мусора определяет, может ли значение быть очищено, он подсчитывает не только имена, но и все другие ссылки.

Обратите внимание, что `i = x` — это присваивание значения `x` имени `i`. Но в примере `i[0] = x` значение присваивается не имени `i`. Оно присваивается первому элементу значения `i`. Очень важно понимать, чему именно присваивается значение. Тот факт, что какое-то имя появляется в левой части выражения присваивания, еще не означает, что именно это имя получило новое значение.

…и многие сущности могут быть значениями

Так же, как многие вещи могут служить в качестве ссылок, многие операции являются значениями. Каждая из следующих строк — значение для имени `X`:

```
X = ...
for X in ...
[... for X in ...]
(... for X in ...)
{... for X in ...}
class X(...):
def X(...):
def fn(X): ... ; fn(12)
with ... as X:
except ... as X:
import X
from ... import X
import ... as X
from ... import ... as X
```

Это не значит, что эти предложения действуют, как присваивание: они *являются* присваиваниями. Все они делают имя `x` ссылкой на значение. И все, что до сих пор говорилось присваиваниях, касается и этих случаев.

Большинство из этих предложений определяют `x` в области видимости самого предложения, но не все (особенно это касается представлений). Кроме того, есть некоторые отличия в деталях между Python 2 и Python 3. Тем не менее, все они — настоящие присваивания, и все факты, касающиеся присваиваний, касаются и этих предложений.

Циклы for

Циклы `for` — любопытный пример. Когда вы пишете следующий код:

```
for x in sequence:
    something(x)
```

он выполняется примерно так:

```
x = sequence[0]
something(x)
x = sequence[1]
something(x)
# и так далее...
```

На самом деле механизм получения значений из последовательности более сложен, чем простая индексация в примере. Но суть в том, что каждый элемент в последовательности присваивается имени `x` в качестве значения, как если бы это делалось путем простого оператора присваивания. И, опять-таки, все правила, касающиеся присваиваний и их работы, применимы и к этому присваиванию.

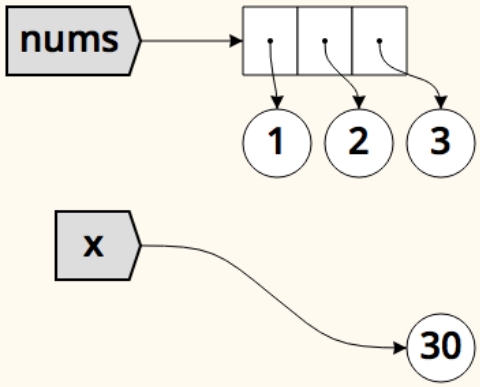
Допустим, у нас есть список чисел. Мы хотим умножить каждое число на 10, чтобы из [1, 2, 3] получить [10, 20, 30]. Можно попытаться пойти простым путем, но нас ждет разочарование.

```
nums = [1, 2, 3]
for x in nums: # x = nums[0] ...
    x = x * 10
print(nums) # [1, 2, 3] :(
```

Давайте разберемся, почему так. При первой итерации `x` — это еще одно имя для `nums[0]`. Как мы разбирали ранее, если у вас есть два имени, ссылающиеся на одно значение, то при повторном присвоении значения одному из имен второе не меняется вместе с ним. В данном случае мы дали имени `x` новое значение (`x = x * 10`), так что `x` теперь ссылается на 10. Но `nums[0]` по-прежнему ссылается на старое значение, 1.

Наш цикл не изменит исходный список, потому что мы просто раз за разом присваиваем новое значение для имени `x`.

```
nums = [1, 2, 3]
for x in nums:          # x = nums[0] ...
    x = x * 10
print(nums)             # [1, 2, 3]   :(
```



Лучшее, что можно посоветовать в такой ситуации, — это не изменять списки, а создавать новые:

```
nums = [ 10*x for x in nums ]
```

Аргументы функций — это присваивания

Аргументы функций — это, пожалуй, самая важная вещь из тех, что на первый взгляд не кажутся присваиваниями, хотя и являются ими. При определении функции вы указываете ее формальные параметры. В следующем примере это `x`:

```
def func(x):
print(x)
```

При вызове функции вы передаете в нее значения настоящих аргументов:

```
num = 17
func(num)
print(num)
```

Здесь `num` — это значение, замещающее параметр `x`. При вызове этой функции мы получаем точно такое же поведение, как если бы выполнили `x = num`. Параметру присваивается настоящее значение.

Каждый вызов функции создает фрейм стека — контейнер для имен, локальных для этой функции. Имя `x` — локально для функции, но суть присваивания остается прежней.

Когда мы находимся внутри функции, у нас есть значение (17) с двумя именами: `num` в вызывающем фрейме и `x` — во фрейме функции.

Когда функция делает возврат значения, фрейм стека функции уничтожается, а вместе с ним — содержащиеся в нем имена. Эти имена могут возвращать (но это не обязательно) значения, если они были последними именами, ссылающимися на эти значения.

Давайте попробуем написать полезную функцию. Мы хотим дважды добавить значение в список. напишем это тремя разными способами. Два из них будут рабочими (хотя работать будут по-разному), а один будет совершенно бесполезным. Он нам нужен лишь для иллюстрации того, почему три варианта кода ведут себя по-разному.

Вариант 1

Рассмотрим первый вариант написания функции `append_twice`:

```
def append_twice(a_list, val):
    a_list.append(val)
    a_list.append(val)
```

Тут все очень просто, функция делает именно то, что заявлено в ее имени.

Вызов происходит так:

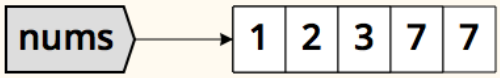
```
nums = [1, 2, 3]
append_twice(nums, 7)
print(nums) # [1, 2, 3, 7, 7]
```

Вызывая функцию `append_twice`, мы передаем ей список `nums`. Таким образом `nums` присваивается в качестве значения параметру `a_list`. В результате у нас есть два имени для одного списка: `nums` во фрейме вызова и `a_list` во фрейме функции `append_twice`. После этого мы дважды добавляем `val` в конец списка. Операции добавления осуществляются со списком `a_list`, а это тот же список, что и `nums` во фрейме вызова. Таким образом, мы напрямую модифицируем список во фрейме вызова.

Когда работа функции завершается, фрейм уничтожается и таким образом удаляется локальное имя `a_list`. Поскольку это была не единственная ссылка на этот список, удаление имени `a_list` не стирает сам список.

Во фрейме вызова мы выводим на экран список `nums` и видим, что он действительно бы изменен.

```
def append_twice(a_list, val):
    a_list.append(val)
    a_list.append(val)
    return

nums = [1, 2, 3]
append_twice(nums, 7)
print(nums)          # [1, 2, 3, 7, 7]
```



А теперь давайте попробуем реализовать это иначе.

Вариант 2

```
def append_twice_bad(a_list, val):
    a_list = a_list + [val, val]
    return

nums = [1, 2, 3]
append_twice_bad(nums, 7)
print(nums) # [1, 2, 3]
```

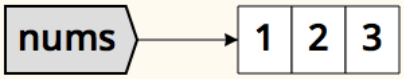
Здесь используется другой подход: мы расширяем список внутри функции, добавляя два значения к исходному списку. Но эта функция вообще не работает. Как и раньше, мы передаем в нее `nums`, так что и `nums`, и `a_list` начинают ссылаться

на исходный список. Но затем мы создаем совершенно новый список, присваивая его в виде значения для `a_list`.

Когда работа функции завершается, фрейм уничтожается, а вместе с ним и имя `a_list`. Но на наш новосозданный список ссылалось только одно имя — `a_list`, и с уничтожением имени список тоже стирается. Вся наша работа потеряна!

Возвращаясь ко фрейму вызова, мы не видим вообще никакого эффекта, потому что исходный список не был изменен.

```
def append_twice_bad(a_list, val):
    a_list = a_list + [val, val]
    return

nums = [1, 2, 3]
append_twice_bad(nums, 7)
print(nums)          # [1, 2, 3]
```



Мораль: функции, которые изменяют передаваемые в них значения, — отличный способ получить неожиданные баги. Создание новых списков поможет избежать подобных проблем.

К счастью, исправить нашу функцию совсем не сложно.

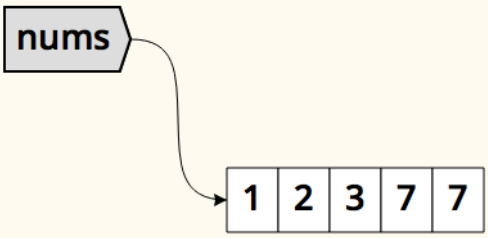
Вариант 3

Здесь у нас практически все то же самое, но, создав новый список `a_list`, мы возвращаем его:

```
def append_twice_good(a_list, val):
a_list = a_list + [val, val]
return a_list

nums = [1, 2, 3]
nums = append_twice_good(nums, 7)
print(nums) # [1, 2, 3, 7, 7]
```

Затем, во фрейме вызова, мы не просто вызываем `append_twice_good`, мы присваиваем возвращенное ею значение списку `nums`. Функция создает новый список. Возвращая его, мы можем определить, что с ним делать. В нашем случае мы хотим, чтобы `nums` стал новым списком, так что мы присваиваем это значение имени `nums`. Старое значение стирается, а новое остается.

```
def append_twice_good(a_list, val):
    a_list = a_list + [val, val]
    return a_list

nums = [1, 2, 3]
nums = append_twice_good(nums, 7)
print(nums)          # [1, 2, 3, 7, 7]
```



Ниже представлены все три версии этой функции — для сравнения. В первом варианте функция изменяет исходный список. Во втором она создает новый список, но мы никак не можем воспользоваться результатом. В третьем варианте функция создает новый список и возвращает его, чтобы мы могли его использовать.

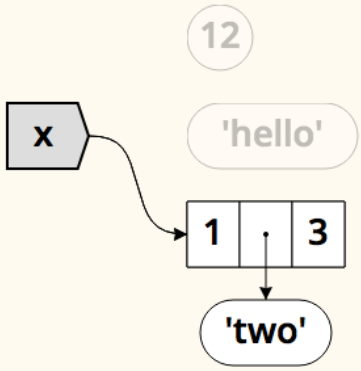
```
def append_twice(a_list, val):
    """Изменяет аргумент"""
    a_list.append(val)
    a_list.append(val)

def append_twice_bad(a_list, val):
    """Бесполезная функция"""
    a_list = a_list + [val, val]
    return

def append_twice_good(a_list, val):
```

```
    """Возвращает новый список"""
    a_list = a_list + [val, val]
    return a_list
```

Любое имя может ссылаться на любое значение

Python — язык с динамической типизацией. Это означает, что имена не имеют типа. Любое имя в любой момент может начать ссылаться на любое значение. Имя может ссылаться на целое число, а затем — на строку, а после этого — на функцию, а потом — на модуль. Конечно, такая программа будет очень запутанной — так делать не надо! Но Python это допускает.

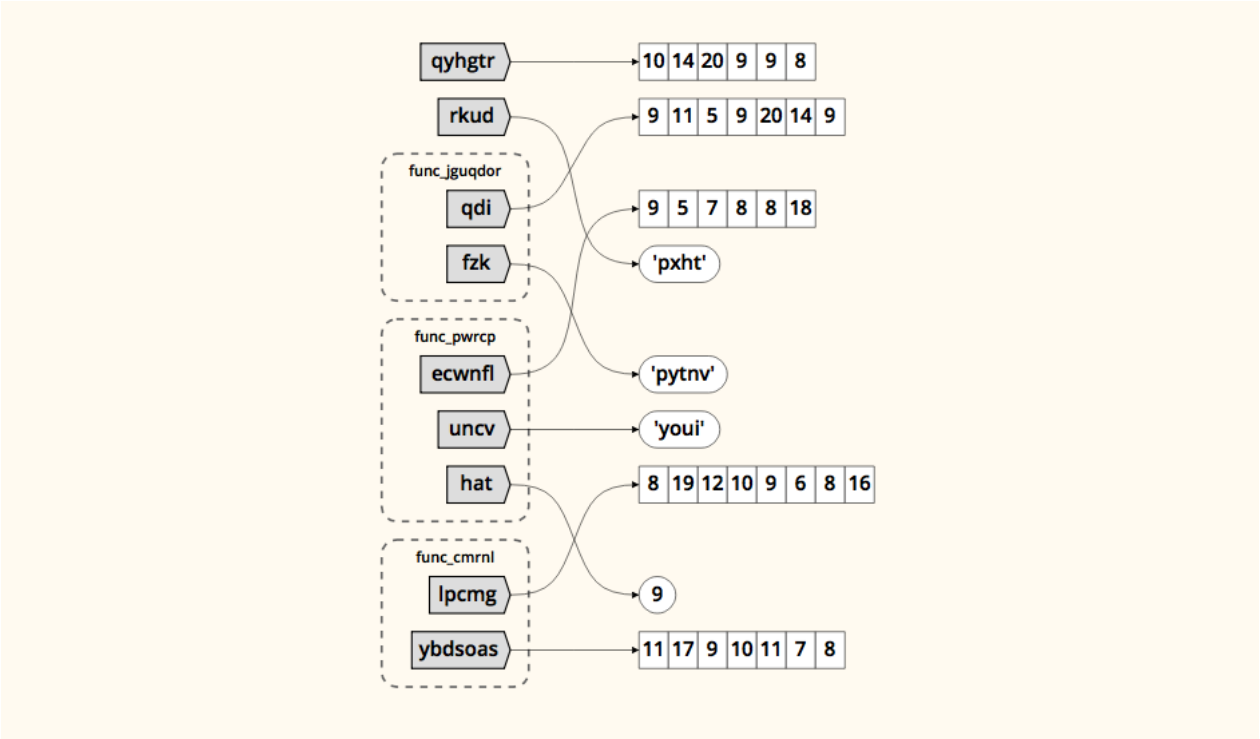
```
x = 12
x = "hello"
x = [1, 2, 3]
x[1] = "two"
```



Динамическая типизация

Имена не имеют типов, значения не имеют областей видимости

Так же, как имена не имеют типов, значения не имеют областей видимости. Когда мы говорим, что функция имеет локальную переменную, мы имеем в виду, что *имя* этой переменной принадлежит к области видимости этой функции. Вы не можете использовать это имя вне этой функции. После того как функция сделает возврат, имя будет стерто. Но, как мы видели, если значение, на которое ссылалось это имя, имеет и другие имена, оно останется и вне вызова функции. Локально имя, а не значение.



Значения могут создаваться, использоваться и, в конечном итоге, стираться в стеке вызова. Нет никаких правил относительно того, когда могут создаваться значения и сколько они могут жить.

Дополнительная информация

Давайте разберем еще несколько небольших вопросов.

Python не имеет переменных

Объясняя, как работает Python, люди часто упоминают, что он не имеет переменных. Это, разумеется, не так. Имеется в виду, что переменные в Python работают иначе, чем в C. Впрочем, даже это утверждение трудно понять.

Когда-то языку Python учились люди, знающие язык C. Теперь все изменилось. Но даже если бы мы имели дело с массовыми перебежчиками из лагеря C, совершенно незачем отдавать языку C права на слово «переменная», вынуждая питонистов пользоваться какими-то другими словами. Два разных языка вполне могут использовать одно слово по-разному — это нормально.

Вызов по значению или по ссылке?

Пытаясь объяснить, использует Python вызов по ссылке или по значению, люди частенько путаются. На самом деле это ложная дихотомия. Она порождена попыткой неправильно применить концепцию одного языка во всех остальных.

Python использует соглашение о вызове несколько иначе, у него нет ни вызова по ссылке, ни вызова по значению в чистом виде. Некоторые называют это вызовом по объекту или вызовом по присваиванию.

Создание 2D-списков

Простой способ создания вложенного списка (скажем, шахматку 8×8) не работает.

```
board = [[0] * 8] * 8    # плохо
```

Вы не получите восемь отдельных списков с нулями. Это будет внутренний список из восьми нулей и внешний список, содержащий восемь ссылок на внутренний.

```
board = [[0] * 8 for _ in range(8)]    # хорошо
```

А второй способ даст вам нужную структуру. Если вам интересна эта тема, можно почитать статью «Names and values: making a game board».