


Понимание работы логических операций and, or и not



mango-office.ru

РЕКЛАМА

Виртуальная АТС Расширенная

2 000 ₽

Узнать больше

[Справочник по языку Python3.](#) / Понимание работы логических операций and, or и not

Синтаксис:

```
x or y
x and y
not x
x and y and not z
x and (y or z)
```

Важно! Операторы and и or закорачивают вычисление своих операндов (т.е. используют замыкания): правый операнд вычисляется лишь в том случае, если его значение необходимо для получения истинного значения в операциях and или or. Другими словами, замыкания в логических операциях используются для запуска второй части или последующих частей логического выражения только в том случае, если это актуально!

- or - оценивает второй аргумент, только если первый равен False. Если какой либо операнд в цепочке or является истиной, немедленно возвращается результат - первое истинное значение.
- and - оценивает второй аргумент, только если первый равен True. Если в цепочке and все операнды являются истиной, результатом будет последнее значение. А если какой-либо из операндов является False, результатом будет первое ложное значение.
- not имеет более низкий приоритет, чем [операторы сравнения](#), так not a == b интерпретируется как not (a == b), а выражение a == not b вовсе является синтаксической ошибкой. Единственный логический оператор с одним аргументом. Он принимает один аргумент x и возвращает противоположный результат: False для истинного значения и True для ложного значения.

Операторы and и or не приводят свои результаты принудительно к значениям True или False, а возвращают один из своих операндов. Такой подход позволяет использовать эти операторы в более общих, а не только булевых операциях. Если другие операторы, прежде чем выполнить операцию, вычисляют все свои операнды, то в случае операторов and и or с их семантикой закорачивания необходимость вычисления правого операнда определяется результатом вычисления левого.

Из булевых операторов, not имеет самый высокий приоритет, а or самый низкий, так что A and not B or C эквивалентно (A and (not B)) or C. Как всегда, скобки могут быть использованы для выражения желаемого приоритета в операциях.

[Логические операции](#), упорядоченные по приоритету выполнения:

- not x - если x ложно, то возвращается True, иначе False.
- x and y - если x ложно, то возвращается x, иначе y.
- x or y - если x ложно, то возвращается y, иначе x

Объяснение работы замыкания с оператором and:

```
a = 'a'
b = 'b'
c = 'c'

>>> a and b
# 'b'

>>> '' and b
# ''
```

```
>>> a and b and c
# 'c'
```

Пояснения к примеру выше с оператором and:

- 1. Оператор and вычисляет значения в булевом контексте слева направо. Значения 0, '', [], (), {} и None являются ложью, все остальное является истиной. Если у and оба операнда являются истиной, результатом будет последнее значение.
- 2. Если какой-либо из операндов является ложью, результатом будет первое такое значение. В данном случае это '' - пустая строка, первое значение которое является ложью.
- 3. Все значения являются истиной, так что в результате мы получаем последнее c.

Объяснение работы замыкания с оператором or:

```
a = 'a'
b = 'b'

>>> a or b
# 'a'

>>> '' or b
# 'b'

>>> '' or [] or {}
# {}

>>> def func():
...     return 1
>>> a or func()
# 'a'
```

Пояснения к примеру выше с оператором or:

- 1. Оператор or вычисляет значения в булевом контексте слева направо. Если операнд является истиной, or немедленно возвращает результат. В данном случае a, первое истинное значение.
- 2. or вычисляет выражение '', которое является ложью, затем b, которое является истиной, и возвращает его значение.
- 3. Если все значения являются ложью, or возвращает последнее.
- 4. Обратите внимание, что or вычисляет операнды до тех пор, пока не найдет истинное значение, остальное игнорируется. Это имеет значение, когда вычисление операнда дает сторонние эффекты. В данном случае функция func() не вызывается, так как для получения результата выражения с оператором or достаточно того, что первый операнд a является истиной.

Другие примеры с and и or:

```
>>> a = 'one'
>>> b = 'two'
>>> 1 and a or b
# 'one'
>>> 0 and a or b
# 'two'

>>> a = ''
>>> b = 'two'
# 'a' - пустая строка, которую Python считает ложью,
# следовательно 1 and '' дает '', a '' or 'two' дает 'two'.
>>> 1 and a or b
# 'two'
```

Практическое использование замыканий and и or в кодовой базе.

Внимание! Замыкания в операциях and и or можно использовать с пользой в вычислениях для экономии ресурсов, сокращения времени выполнения и т.д., только будьте осторожны! Необходимо четко понимать как работают замыкания в операторах and и or.

- [Экономия ресурсов и времени выполнения при помощи and](#);
- [Проверка предварительных условий перед выражением](#);
- [Определение значения по умолчанию при помощи or](#);
- [Пример использования замыканий в функциях all\(\) и any\(\)](#).

Экономия ресурсов и времени выполнения при помощи and.

Рассмотрим реальный пример из [модуля base64](#) стандартной библиотеки Python, который использует замыкание в [оператора if](#). Исследуем функцию b64decode, которая берет строку (или объект, подобный байтам) и декодирует ее:

```
# взято из Lib/base64.py
def b64decode(s, altchars=None, validate=False):
    """Decode the Base64 encoded bytes-like object or ASCII string s.
    [docstring cut for brevity]
    """
    s = _bytes_from_decode_data(s)
    if altchars is not None:
        altchars = _bytes_from_decode_data(altchars)
        assert len(altchars) == 2, repr(altchars)
        s = s.translate(bytes.maketrans(altchars, b'+/'))
    # использование замыкания с оператором `and`
    if validate and not re.fullmatch(b'[A-Za-z0-9+/]*={0,2}', s):
        raise binascii.Error('Non-base64 digit found')
    return binascii.a2b_base64(s)
```

Смотрим на оператор if, который помечен комментарием. В условии сначала проверяется аргумент validate, а только потом результат работы функции re.fullmatch(). Аргумент validate сообщает функции, хочет ли пользователь вообще проверять строку, которую нужно декодировать. Обратите внимание, что если validate=False то сопоставление регулярного выражения не запускается (срабатывает замыкание). Если порядок операндов поменять, то результат остался такой же, но было бы потрачено гораздо больше времени.

Проверка предварительных условий перед выражением.

Другой типичный шаблон использования замыканий проявляется, когда перед определенной операцией, которая может вызвать исключение, нужно что-то проверить.

Допустим есть последовательность и нужно взять элемент по индексу, но последовательность может оказаться пустой или ее длина может быть меньше индекса. Например:

```
>>> lst = [0, 1, 2]
>>> lst[3]
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
# IndexError: list index out of range
```

Используем проверку предварительных условий:

```
>>> lst = [0, 1, 2]
>>> if lst and len(lst) >= 3:
...     lst[3]
```

Здесь, в условии сначала проверяется что список НЕ пустой и только после этого вычисляется длинна этого списка. Если список пустой, то его длина проверяться не будет и условие if завершиться.

Еще один пример из [модуля enum](#):

```
# взято из Lib/enum.py
def _create(cls, class_name, names, *, module=None, qualname=None, type=None, start=1):
    """
    Convenience method to create a new Enum class.
    """
    # [сокращено для краткости]

    # special processing needed for names?
    if isinstance(names, str):
        names = names.replace(',', ' ').split()
```

```
# смотрим на следующие условие
if isinstance(names, (tuple, list)) and names and isinstance(names[0], str):
    original_names, names = names, []
    last_values = []

# [сокращено для краткости]
```

Более длинный оператор if содержит три выражения, разделенных операторами and, и первые два выражения нужны для того, чтобы убедиться, можно ли безопасно выполнить последнее.

- `isinstance(names, (tuple, list))` - проверяет, является ли `names` кортежем или списком. Если не является то условие завершается.
- далее `names` проверяется пусто оно или нет. Если элементов нет то условие завершается.
- если `names` не пустой, то можно безопасно выполнить последнюю проверку, связанную с индексацией `names[0]`, а именно `isinstance(names[0], str)`.

Определение значения по умолчанию при помощи or.

Замыкание с помощью логического оператора or может использоваться для присвоения переменным значений по умолчанию. Вот пример:

```
# test.py
greet = input("Ваше имя >> ") or "незнакомец"
print(f"Привет, {greet}!")

# $python3 -i test.py
# Ваше имя >>
# Привет, незнакомец!
```

Если запустить этот пример и ничего не вводя нажать Enter, то получим вывод "Привет, незнакомец!". Что тут происходит? Если ничего не вводить и нажать Enter, то [функция input\(\)](#) вернет пустую строку '', что является False. Следовательно, оператор or видит ложное значение слева и должен оценить правый операнд. Для определения окончательное значение выражения or оценивает правый операнд и если он True, то возвращает его значение.

Еще пример присвоения значение по умолчанию (используя or) для изменяемого аргумента из стандартной библиотеки Python.

```
# взято из Lib/cgitb.py
class Hook:
    """A hook to replace sys.excepthook that shows tracebacks in HTML."""

    def __init__(self, display=1, logdir=None, context=5, file=None,
                 format="html"):
        self.display = display          # send tracebacks to browser if true
        self.logdir = logdir            # log tracebacks to files if not None
        self.context = context          # number of source code lines per frame
        self.file = file or sys.stdout  # <= смотрите сюда
        self.format = format
```

Этот код взят из модуля cgitb и определяет [sys.stdout](#) как значение по умолчанию для переменной `self.file`. Определение функции `__init__` имеет `file=None` в качестве ключевого аргумента, так почему бы просто не написать `file=sys.stdout`?

Проблема в том, что `sys.stdout` может быть изменяемым объектом, поэтому использование `file=sys.stdout` в качестве ключевого аргумента со значением по умолчанию не будет работать так, как ожидается. Это легче продемонстрировать со списком в качестве аргумента по умолчанию, хотя принцип тот же:

```
>>> def addend(val, l=[]):
...     l.append(val)
...     print(l)

>>> addend(3, [1, 2])
# [1, 2, 3]
>>> addend(5)
# [5]
>>> addend(5)
```

```
# [5, 5]
>>> addend(5)
# [5, 5, 5]
```

Обратите внимание на три последовательных вызова addend(5). Ожидается, что вызов addend(5) со значением по умолчанию l=[] будет вести себя одинаково, но т.к. список является изменяемым объектом, то вызовы добавляют значения val к значению по умолчанию [], при этом список растет! Дополнительно смотрите материал "[Список Python как аргумент по умолчанию](#)".

Пример использования замыканий or и and в функциях all() и any().

Если в выражении генератора использовать [оператор моржа :=](#), и принимать во внимание тот факт, что функции [all\(\)](#) и [any\(\)](#) также используют замыкания, то можно использовать следующую конструкцию для извлечения первого истинного элемента.

```
any(predicate(witness := item) for item in items)
```

Другими словами, если какой-либо элемент item удовлетворяет условию в функции predicate(), то переменная witness сохранит его значение!

Например, если последовательность содержит много целых чисел, как выяснить, есть ли там какие-либо нечетные числа, и как вывести первое из них?

```
items = [14, 16, 18, 20, 35, 41, 100]
any_found = False
for item in items:
    any_found = item % 2
    if any_found:
        print(f"Найдено нечетное число {item}.")
        break
```

Теперь все это сравним со следующим кодом:

```
>>> items = [14, 16, 18, 20, 35, 41, 100]
>>> is_odd = lambda x: x % 2
>>> if any(is_odd(witness := item) for item in items):
...     print(f"Найдено нечетное число {witness}.")

# Найдено нечетное число 35.
```

ХОЧУ ПОМОЧЬ
ПРОЕКТУ

РЕКЛАМА

FIRECODE

PYTHON

Django - DRF - Flask

firecode.ru

Аутсорсинг
разработки
на Python - Django,
DRF, Flask

4,7

★ Рейтинг организации

Обслуживаем крупнейшие
компании РФ: FinTech, MedTech,
Retail, EdTech, AgroTech

Обсудить задачи >

Большие команды >

Узнать больше