Распаковка и запаковка значений

В конце урока про возврат значений из функции, мы коснулись темы возврата нескольких значений и множественного присваивания получившихся значений нескольким переменным.

def get_coordinates(): return 1, 2

x, y = get_coordinates() print(x) # => 1 print(y) # => 2

Хотя этот прием с присваиванием результата нескольким значениям часто используется именно в применении к функциям, на самом деле, никакого отношения к механике работы функций он не имеет. В приведенном примере функция просто возвращает кортеж, а всю дальнейшую работу делает механизм множественного присваивания, а точнее процедура «распаковывания». Вы с ней уже сталкивались, когда обсуждали кортежи. Сейчас мы посмотрим на возможности множественного присваивания внимательнее.

In [3]:
```
def get_coordinates():
    return 1, 2


x, y = get_coordinates()
z = get_coordinates()
print(x) # => 1
print(y) # => 2
print(z)
```

```
1
2
(1, 2)
```

Итак, вы можете написать: x, y = (1.5, 2.5) В момент присваивания кортеж будет распакован, его первый элемент будет записан в x, второй — в y. Распаковать можно не только кортежи, но и списки: x, y = [1.5, 2.5] будет работать точно так же.

In [16]:
```
# несколько значений в одну переменную. Это делается при помощи звёздочки перед именем переменной:
x, *y, rest = 1, 2, 3, 4, 5, 6, 7, 8
print(x)
print(y)
print(rest)
```

```
1
[2, 3, 4, 5, 6, 7]
8
```

In [9]:
```
# Учтите, что rest всегда будет списком, даже когда в него попадает лишь один элемент или даже ноль:

x, y, *rest = 1, 2, 3
print(rest)
# => [3]
```

```
[3]
```

In [2]:
```
#Звездочка может быть только у одного аргумента, но не обязательно у последнего:

*names, surname = 'Анна Мария Луиза Медичи'.split()
print(names)   # => ['Анна', 'Мария', 'Луиза']
print(surname) # => Медичи
```

```
['Анна', 'Мария', 'Луиза']
Медичи
```

In [17]:
```
# посередине

li = ["Дейнерис",
      'королева андалов, ройнаров и Первых Людей',
      'королева Миэрина',
      'кхалиси Великого Травяного моря',
      'Неопалимая',
      'Бурерожденная',
      'Матерь Драконов',
      'Разрушительница Оков',
      'Низвергательница Колдунов',
      'Таргариен']
name, *titles, surname = li
print(name)
print(titles)
print(surname)
```

```
Дейнерис
['королева андалов, ройнаров и Первых Людей', 'королева Миэрина', 'кхалиси Великого Травяного моря', 'Неопалимая', 'Бурерожденн
ая', 'Матерь Драконов', 'Разрушительница Оков', 'Низвергательница Колдунов']
Таргариен
```

In [21]:
```
# Также есть возможность распаковывать вложенные списки. Например
c = 0
l1 = [2,3]
a, (b, c), d = [1, l1, 4]
# a, b, d = [1, l1, 4]

print(a)
print(b)
print(c)
print(d)
```

```
1
2
3
4
```

In [6]:
```
#Если вы хотите распаковать единственное значение в кортеже, то после имени переменной должна идти запятая:

a = (1,)
b, = (1,)
print(a)  # => (1,)
print(b)  # => 1
```

```
(1,)
1
```

In [7]:
```
#Помимо распаковывания есть и операция запаковывания. Она выполняется всегда, когда справа от знака равенства стоит
# больше одного значения. Например, можно написать: values = 1, 2, 3
#Тогда значения в правой части автоматически будут запакованы в кортеж (1, 2, 3).
#Запаковывание можно комбинировать с распаковыванием:

a, *b = 1, 2, 3
print(a, b)
# => 1 [2, 3]
```

```
1 [2, 3]
```

Техника запаковывания и распаковывания переменных со звездочкой используется не только в операциях присваивания. Похожий механизм применяется для написания функций, которые могут принимать переменное число аргументов. И синтаксис для этого используется похожий на тот, который используется в множественном присваивании. В списке аргументов функции, один из аргументов может быть помечен звездочкой - тогда в него попадут все значения на соответствующей позиции, которые еще не присвоены другим аргументам.

Но есть и отличие. При множественном присваивании переменная со звездочкой получает список значений. А когда аргумент функции указан со звездочкой, он получает кортеж значений.

Разработчики языка Python могли поступить иначе и сделать функцию, принимающую всегда ровно один аргумент-список, и выводить на экран элементы этого списка. С точки зрения функциональности результат был бы аналогичным, но такую функцию было бы не так удобно использовать.

In [25]:
```
# Функция произведения неограниченного кол-ва аргументов
def product(first, *rest):
    result = first
    for value in rest:
        result *= value
    return result


print(product(2,3,5,7,9))
```

```
2
```

In [9]:
```
arr = ['cd', 'ef', 'gh']
# Здесь мы передаем просто список как один аргумент
print(arr) # => ['cd', 'ef', 'gh']

# А здесь мы раскрыли список и
#   функция print получила три отдельных аргумента
print(*arr) # => cd ef gh
# Это аналогично вызову
print('cd', 'ef', 'gh') # => cd ef gh

# Раскрытие списка можно комбинировать с любыми аргументами
print('ab', *arr, 'yz') # => ab cd ef gh yz
# При раскрытии может быть несколько аргументов со звездочкой
print(*arr, *arr) # => cd ef gh cd ef gh
```

```
['cd', 'ef', 'gh']
cd ef gh
cd ef gh
ab cd ef gh yz
cd ef gh cd ef gh
```

20.12.2020 Группа 2.

Аргументы по умолчанию

Бывает так, что какой-то параметр функции часто принимает одно и то же значение. Например, хорошо известная вам функция int принимает два параметра: строка, которую нужно преобразовать в число, а также основание системы счисления. Это позволяет ей считывать числа в различных системах счисления, например, двоичное число 101 мы можем считать так: int('101', 2) # => 5

In [11]:
```
#В качестве примера сделаем функцию, которая будет готовить бургеры с котлетами разного типа
# и по умолчанию добавлять туда помидоры, но не добавлять лук. Тогда функция приготовления будет выглядеть так:

def make_burger(type_of_meat, with_onion=False, with_tomato=True):
    print('Булочка')
    if with_onion:
        print('Луковые колечки')
    if with_tomato:
        print('Ломтик помидора')
    print('Котлета из', type_of_meat)
    print('Булочка')
```

Первыми стоит указывать более важные аргументы (в нашем примере мы считаем, что класть или не класть лук — более важное решение, чем класть или не класть помидор). Если вы укажете только одно дополнительное значение, то оно будет присвоено первому аргументу по умолчанию, а второй аргумент так и останется со значением по умолчанию. Если укажете два значения, то значения будут присвоены обеим переменным.

Аргументы, которые передаются без указания имён, называются позиционными, потому что функция по положению аргумента понимает, какому параметру он соответствует. Аргументы, которые передаются с именами, называются именованными. Чтобы вашу функцию можно было вызывать, используя именованные аргументы, буквально ничего не нужно делать. Все функции, которые вы писали на предыдущих уроках, уже можно вызывать, передавая им именованные аргументы.

In [13]:
```
#Именованные аргументы можно использовать вместе со значениями по умолчанию. Например,
#мы можем вызвать нашу функцию для создания бургеров, передав ей нужные именованные аргументы,
#а остальные оставив значениями по умолчанию (так как мы используем именованные аргументы,
#нам теперь не важно, в каком порядке мы их определяли):
make_burger(type_of_meat='говядина', with_tomato=False)
```

```
Булочка
Ломтик помидора
Котлета из говядина
Булочка
```

Именованные и позиционные аргументы не всегда хорошо друг с другом "ладят". При вызове функции позиционные аргументы должны обязательно идти перед именованными аргументами. Достаточно сложно сформулировать точные правила поведения аргументов функции при использовании одновременно аргументов со звездочкой и именованных аргументов. Чем запоминать точные правила в таких случаях лучше пользоваться здравым смыслом.

Общие рекомендации следующие: Если вам приходится долго думать о том, как записать список аргументов, чтобы он работал корректно, лучше использовать более простую версию. Ведь код, который тяжело писать, с большой вероятностью будет тяжело читать.

Если ваш вызов функции не работает, попробуйте прочитать его "глазами интерпретатора". Однозначно ли он читается или вы можете придумать несколько вариантов разложить переданные в вызове функции параметры по аргументам? Если вы можете трактовать код несколькими способами, то с большой вероятностью, интерпретатор столкнется с теми же трудностями. В ситуации, когда код неоднозначен, интерпретатор Python не пытается угадать, что программист имел ввиду, а сообщает об ошибке. Часто это считается синтаксической ошибкой, и ошибка возникает еще до того как программа начинает выполняться.

Инструкция pass. Согласованность аргументов В языке Python есть эталонно бесполезная инструкция pass. Инструкция pass - это инструкция-заглушка, которая не делает ничего. Дело в том, что синтаксис языка Python не позволяет в некоторых местах обойтись без команд. Например, не может быть функции с пустым телом. Ветвь условного оператора или тело цикла тоже должны выполнять какие-либо действия, но иногда программист хочет отложить их написание и ставит такую заглушку.

In [ ]:
```
if game_is_over:
    pass # TODO: написать вывод итогового результата
```

In [ ]:
```
#Чтобы написать функцию, которая игнорирует любой список аргументов,
#необходимо разрешить ей принимать произвольное число позиционных аргументов и произвольное число именованных аргументов:

def nop(*rest, **kwargs):
    pass

nop(1,[2, 3], debug=True, file="debug.log")
```

In [16]:
```
#Аргумент с двумя звёздочками, **kwargs - это специальный аргумент, который может перехватить все "лишние" именованные аргумент
ы,
#переданные в функцию. Лишними аргументами будут все именованные аргументы в команде вызова функции,
#для которых нет соответствующего параметра в определении функции.

#Этот аргумент, как и аргумент с одной звёздочкой, захватывающий "лишние" позиционные аргументы,
#можно использовать в комбинации с обычными аргументами. Например, сделаем и вызовем функцию,
#распечатывающую информацию о пользователе:

def profile(name, surname, city, *children, **additional_info):
    print("Имя:", name)
    print("Фамилия:", surname)
    print("Город проживания:", city)
    if len(children) > 0:
        print("Дети:", ", ".join(children))
    print(additional_info)


profile("Сергей", "Михалков", "Москва", "Никита Михалков",
        "Андрей Кончаловский", nnn="Самолет", occupation="writer", diedIn=2009)
```

```
Имя: Сергей
Фамилия: Михалков
Город проживания: Москва
Дети: Никита Михалков, Андрей Кончаловский
{'nnn': 'Самолет', 'occupation': 'writer', 'diedIn': 2009}
```

Как вы уже знаете, параметр children будет списком лишних позиционных аргументов. А вот additional_info будет словарём лишних именованных аргументов. В последней строке мы распечатали переданный словарь, он выглядит так: {'diedIn': 2009, 'occupation': 'writer'}