Функции

Необязательные и именованные аргументы

Позиционные аргументы

Все ранее написанные нами функции имели **позиционные аргументы**. Такие аргументы передаются без указания имен. Они называются **позиционными**, потому что именно по позиции, расположению аргумента, функция понимает, какому параметру он соответствует.

Рассмотрим следующий код:

```
def diff(x, y):
    return x - y

res = diff(10, 3)    # используем позиционные аргументы
print(res)
```

Такой код выведет число 7. При вызове функции `diff()` первому параметру `x` будет соответствовать первый переданный аргумент, 10, а второму параметру `y` — второй аргумент, 3.

В Python можно использовать не только позиционные, но и именованные аргументы.

Именованные аргументы

Аргументы, передаваемые с именами, называются **именованными**. При вызове функции можно использовать имена параметров из ее определения. Исключение составляют списки аргументов неопределенной длины, где используются аргументы со звездочкой, но об этом в следующем уроке. Все функции из предыдущих уроков можно вызывать, передавая им именованные аргументы.

Рассмотрим следующий код:

```
def diff(x, y):
    return x - y

res = diff(x=10, y=3)   # используем именованные аргументы
print(res)
```

Такой код по-прежнему выведет число 7. При вызове функции `diff()` мы явно указываем, что параметру `x` соответствует аргумент 10, а параметру `y` — аргумент 3.

Использование именованных аргументов позволяет нарушать их позиционный порядок при вызове функции. Порядок упоминания именованных аргументов не имеет значения!

Мы можем вызвать функцию `diff()` так:

```
res = diff(y=3, x=10)
```

и получить тот же результат 7.

Возможность использования именованных аргументов — еще один повод давать параметрам значащие, а не однобуквенные имена.

Когда стоит применять именованные аргументы

Каких-то строгих правил на этот счёт не существует. Однако широко практикуется такой подход: если функция принимает больше трёх аргументов, нужно хотя бы часть из них указать по имени. Особенно важно именовать значения аргументов, когда они относятся к одному типу, ведь без имен очень трудно понять, что делает функция с подобным вызовом.

Рассмотрим определение функции `make_circle()` для рисования круга:

```
def make_circle(x, y, radius, line_width, fill):
    # тело функции
```

Вызвать такую функцию можно так:

```
make_circle(200, 300, 17, 2.5, True)
```

Тут непросто понять, какие параметры круга задают числа 200, 300 или 17.

Сравните:

```
make_circle(x=200, y=300, radius=17, line_width=2.5, fill=True)
```

Такой код читать значительно проще!

В соответствии с PEP 8 при указании значений именованных аргументов при вызове функции знак равенства не окружается пробелами.

Когда значение аргументов очевидно, можно их не именовать. Да, очевидность относительна, но обычно легко понять, что скрывается за значениями при вызове функции `point3d(7, 50, 13)` или `rgb(7, 255, 45)`. В первом случае переданные аргументы – координаты точки в трехмерном пространстве, во втором – значения компонент **red**, **green**, **blue** некоторого цвета. Тут можно ориентироваться только на здравый смысл, жестких правил нет.

Мы уже сталкивались с именованными аргументами, когда вызывали функцию `print()`.

Приведенный ниже код:

```
print('aaaa', 'bbbbb', sep='*', end='##')
print('cccc', 'dddd', sep='()')
print('eeee', 'ffff', sep='123', end='python')
```

использует именованные аргументы `sep` и `end` и выводит:

```
aaaa*bbbbb##cccc()dddd
eeee123ffffpython
```

Используя именованные аргументы `sep` и `end` можно не беспокоиться, какой из них указать первым.

Напомним, что значение по умолчанию для `sep=' '` (символ пробела), а для `end='\n'` (символ перевода строки).

Комбинирование позиционных и именованных аргументов

Мы можем вызывать функции, используя именованные и позиционные аргументы одновременно. Но позиционные значения должны быть указаны **до** любых именованных!

Для функции `diff()` код:

```
res = diff(10, y=3)   # используем позиционный и именованный аргумент
```

работает как полагается, при этом параметру `x` соответствует значение 10.

Приведенный ниже код:

```
res = diff(x=10, 3)   # используем позиционный и именованный аргумент
```

приводит к возникновению ошибки `SyntaxError: positional argument follows keyword argument`.

Необязательные аргументы

Бывает, что какой-то параметр функции часто принимает одно и то же значение. Например, для функции `print()` создатели языка Python установили значения параметров `sep` и `end` равными символу пробела и символу перевода строки, поскольку эти значения используют наиболее часто.

Другим примером служит функция `int()` , преобразующая строку в число. Она принимает два аргумента:

- первый аргумент: строка, которую нужно преобразовать в число;
- второй аргумент: основание системы счисления.

Это позволяет ей считывать числа в различных системах счисления.

Приведенный ниже код, преобразует двоичное число 101:

```
num = int('101', 2)     # аргумент 2 указывает на то, что число 101 записано в двоичной системе
```

Но чаще всего эта функция используется для считывания из строки чисел, записанных в десятичной системе счисления. Утомительно каждый раз писать 10 вторым аргументом. В таких ситуациях Python позволяет задавать некоторым параметрам значения по умолчанию. У функции `int()` второй параметр по умолчанию равен 10, и потому можно вызывать эту функцию с одним аргументом. Значение второго подставится автоматически.

Чтобы задать **значение параметра по умолчанию**, в списке параметров функции достаточно после имени переменной написать знак равенства и нужное значение.

Параметры со значением по умолчанию идут последними, ведь иначе интерпретатор не смог бы понять, какой из аргументов указан, а какой пропущен, и для него нужно использовать значение по умолчанию.

Рассмотрим все то же определение функции `make_circle()`, которая рисует круг:

```
def make_circle(x, y, radius, line_width, fill):
    # тело функции
```

Поскольку обычно нам нужно рисовать круг с шириной линии, равной 1 с заливкой, то логично установить данные значения в качестве значений по умолчанию:

```
def make_circle(x, y, radius, line_width=1, fill=True):
    # тело функции
```

Теперь для того, чтобы нарисовать стандартный круг, то есть круг имеющий ширину линии, равную 1 с заливкой, мы вызываем функцию так:

```
make_circle(100, 50, 20)
```

или так:

```
make_circle(x=100, y=50, radius=20)
```

Если вам хочется поменять ширину линии и заливку, то вы легко можете это сделать:

```
make_circle(x=100, y=50, radius=20, fill=False)                # line_width=1, fill=False
make_circle(x=100, y=50, radius=20, line_width=3)                # fill=True, line_width=3
make_circle(x=100, y=50, radius=20, line_width=5, fill=False)    # line_width=5, fill=False
```

В соответствии с стандартом PEP 8 и при объявлении аргументов по умолчанию пробел вокруг знака равенства не ставят.

Изменяемые типы в качестве значений по умолчанию

При использовании **изменяемых типов данных** в качестве значения параметра по умолчанию можно столкнуться с неожиданными результатами работы функции.

Рассмотрим определение функции `append()`, где в качестве значения по умолчанию используется изменяемый тип данных (список, тип `list`):

```
def append(element, seq=[]):
    seq.append(element)
    return seq
```

Вызывая функцию `append()` следующим образом:

```
print(append(10, [1, 2, 3]))
print(append(5, [1]))
print(append(1, []))
print(append(3, [4, 5]))
```

получим ожидаемый вывод:

```
[1, 2, 3, 10]
[1, 5]
[1]
[4, 5, 3]
```

А если вызовем функцию `append()` так:

```
print(append(10))
print(append(5))
print(append(1))
```

получим **не совсем** ожидаемый вывод:

```
[10]
[10, 5]
[10, 5, 1]
```

Что происходит? Значение по умолчанию для параметра создается единожды при определении функции (обычно при загрузке модуля) и становится атрибутом (свойством) функции. Поэтому, если значение по умолчанию изменяемый объект, то его изменение повлияет на каждый следующий вызов функции.

Чтобы посмотреть значения по умолчанию, можно использовать атрибут `__defaults__`.

Приведенный ниже код:

```
def append(element, seq=[]):
    seq.append(element)
    return seq

print('Значение по умолчанию', append.__defaults__)
```

выводит:

```
Значение по умолчанию ([],)
```

Приведенный ниже код:

```
def append(element, seq=[]):
    seq.append(element)
    return seq

print('Значение по умолчанию', append.__defaults__)
print(append(10))
print('Значение по умолчанию', append.__defaults__)
print(append(5))
print('Значение по умолчанию', append.__defaults__)
print(append(1))
print('Значение по умолчанию', append.__defaults__)
```

выводит:

```
Значение по умолчанию ([],)
[10]
Значение по умолчанию ([10],)
[10, 5]
Значение по умолчанию ([10, 5],)
[10, 5, 1]
Значение по умолчанию ([10, 5, 1],)
```

Для решения проблемы можно использовать константу `None` в качестве значения параметра по умолчанию, а в теле функции устанавливать нужное значение:

```
def append(element, seq=None):
    if seq is None:
```

```
    seq = []
  seq.append(element)
  return seq
```

Вызывая функцию `append()` следующим образом:

```
print(append(10))
print(append(5))
print(append(1))
```

получим **ожидаемый вывод**:

```
[10]
[5]
[1]
```

Подход, основанный на значении `None`, общепринятый в Python.

Примечания

Примечание 1. При написании функций стоит указывать более важные параметры первыми.

Примечание 2. `Именованные аргументы часто используют вместе со значениями по умолчанию.`

Примечание 3. Именованные и позиционные аргументы не всегда хорошо ладят друг с другом. При вызове функции позиционные аргументы должны обязательно идти перед именованными аргументами.

Примечание 4. Параметр в программировании — принятый функцией аргумент. Термин «аргумент» подразумевает, что конкретно и какой конкретной функции было передано, а параметр — в каком качестве функция применила это принятое. То есть вызывающий код передает аргумент в параметр, который определен в описании (заголовке) функции.

- **P**arameter → **P**laceholder (заполнитель принадлежит имени функции и используется в теле функции);
- **A**rgument → **A**ctual value (фактическое значение, которое передается при вызове функции).

Подробнее можно почитать тут.

Примечание 5. Отличная статья на хабре про аргументы функций в Python.

Функции с переменным количеством аргументов

- Аргументы *args
- Аргументы **kwargs
- Keyword-only аргументы

Переменное количество аргументов

Вспомним функцию `print()`, которой мы пользуемся почти в каждой задаче.

Приведенный ниже код:

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

выводит:

```
a
a b
a b c
a b c d
```

Функция `print()` принимает столько аргументов, сколько ей передано. Причем функция `print()` делает полезную работу, даже когда вызывается вообще без аргументов. В этом случае она переносит каретку печати на новую строку.

Было бы здорово научиться писать свои собственные функции, способные принимать переменное количество аргументов. Для этого потребуется специальный, совсем не сложный во всех смыслах, синтаксис.

Рассмотрим определение функции `my_func()`:

```
def my_func(*args):
    print(type(args))
    print(args)


my_func()
my_func(1, 2, 3)
my_func('a', 'b')
```

Приведенный выше код выводит:

```
<class 'tuple'>
()
<class 'tuple'>
(1, 2, 3)
<class 'tuple'>
('a', 'b')
```

В заголовке функции `my_func()` указан всего один параметр `args`, но со звездочкой перед ним. Звездочка в определении функции означает, что переменная (параметр) `args` получит в виде кортежа все аргументы, переданные в функцию при ее вызове от текущей позиции и до конца.

При описании функции можно использовать **только один** параметр помеченный звездочкой, причем располагаться он должен в конце списка параметров, иначе последующим параметрам не достанется значений.

Приведенный ниже код:

```
def my_func(*args, num):
    print(args)
    print(num)
```

не является рабочим, так как параметр со звездочкой указан раньше позиционного `num`.

Приведенный ниже код:

```
def my_func(num, *args):
    print(args)
    print(num)


my_func(17, 'Python', 2, 'C#')
```

связывает с переменной `num` значение 17, а с переменной `args` значение кортежа `('Python', 2, 'C#')` и выводит:

```
('Python', 2, 'C#')
17
```

Помеченный звездочкой параметр `*args` нормально переживает и отсутствие аргументов, в то время как позиционные параметры всегда обязательны.

Приведенный ниже код:

```
my_func(17)
```

связывает с переменной `num` значение 17, а с переменной `args` значение пустого кортежа `()` и выводит:

```
()
17
```

Обратите внимание: функция `my_func()` принимает несколько аргументов, но как минимум один аргумент должен быть передан обязательно. Этот первый аргумент станет значением переменной `num`, а остальные аргументы сохранятся в переменной `args`. Подобным образом можно делать любое количество обязательных аргументов.

Параметр `args` в определении функции пишется после позиционных параметров перед первым параметром со значением по умолчанию.

Передача аргументов в форме списка и кортежа

Иногда хочется сначала сформировать набор аргументов, а потом передать их функции. Тут поможет оператор распаковки коллекций, который также обозначается звездочкой `*`.

Вспомним, что встроенная функция `sum()` принимает на вход коллекцию чисел (список, кортеж, и т.д).

Приведенный ниже код:

```
sum1 = sum([1, 2, 3, 4])       # считаем сумму чисел в списке
sum2 = sum((10, 20, 30, 40))    # считаем сумму чисел в кортеже

print(sum1, sum2)
```

выводит:

```
10 100
```

Однако функция `sum()` не может принимать переменное количество аргументов.

Приведенный ниже код:

```
sum1 = sum(1, 2, 3, 4)

print(sum1)
```

приводит к возникновению ошибки:

```
TypeError: sum expected at most 2 arguments, got 4
```

Напишем свою версию функции `sum()`, функцию `my_sum()`, которая принимает переменное количество аргументов и вычисляет их сумму:

```
def my_sum(*args):
    return sum(args)    # args - это кортеж
```

Приведенный ниже код:

```
print(my_sum())
print(my_sum(1))
print(my_sum(1, 2))
print(my_sum(1, 2, 3))
print(my_sum(1, 2, 3, 4))
```

выводит:

```
0
1
3
6
10
```

Мы также можем вызывать нашу функцию `my_sum()` , передавая ей списки или кортежи, предварительно распаковав их.

Приведенный ниже код:

```
print(my_sum(*[1, 2, 3, 4, 5]))   #  распаковка списка
print(my_sum(*(1, 2, 3)))         #  распаковка кортежа
```

выводит:

```
15
6
```

Более того, часть аргументов можно передавать непосредственно и даже коллекции подставлять не только по одной.

Приведенный ниже код:

```
print(my_sum(1, 2, *[3, 4, 5], *(7, 8, 9), 10))
```

выводит:

```
49
```

По соглашению, параметр, получающий подобный кортеж значений, принято называть `args` (от слова arguments). Старайтесь придерживаться этого соглашения.

Получение именованных аргументов в виде словаря

Позиционные аргументы можно получать в виде `*args`, причём произвольное их количество. Такая возможность существует и для именованных аргументов. Только именованные аргументы получаются в виде словаря, что позволяет сохранить имена аргументов в ключах.

Рассмотрим определение функции `my_func()`:

```
def my_func(**kwargs):
    print(type(kwargs))
    print(kwargs)
```

Приведенный ниже код:

```
my_func()
my_func(a=1, b=2)
my_func(name='Timur', job='Teacher')
```

выводит:

```
<class 'dict'>
{}
<class 'dict'>
{'a': 1, 'b': 2}
<class 'dict'>
{'name': 'Timur', 'job': 'Teacher'}
```

По соглашению параметр, получающий подобный словарь, принято называть `kwargs` (от словосочетания keyword arguments). Старайтесь придерживаться этого соглашения.

Параметр `**kwargs` пишется в самом конце, после последнего аргумента со значением по умолчанию. При этом функция может содержать и `*args` и `**kwargs` параметры.

Рассмотрим определение функции, которая принимает все виды аргументов.

```
def my_func(a, b, *args, name='Gvido', age=17, **kwargs):
    print(a, b)
    print(args)
    print(name, age)
    print(kwargs)
```

Приведенный ниже код:

```
my_func(1, 2, 3, 4, name='Timur', age=28, job='Teacher', language='Python')
my_func(1, 2, name='Timur', age=28, job='Teacher', language='Python')
my_func(1, 2, 3, 4, job='Teacher', language='Python')
```

выводит (пустая строка вставлена для наглядности):

```
1 2
(3, 4)
Timur 28
{'job': 'Teacher', 'language': 'Python'}

1 2
()
Timur 28
{'job': 'Teacher', 'language': 'Python'}

1 2
(3, 4)
Gvido 17
{'job': 'Teacher', 'language': 'Python'}
```

Не нужно пугаться, в реальном коде функции редко используют все эти возможности одновременно. Но понимать, как каждая отдельная форма объявления аргументов работает, и как такие формы сочетать — очень важно.

Для лучшего понимания, поэкспериментируйте с передачей аргументов. Правила использования аргументов довольно сложно описывать, но на практике мы редко сталкиваемся с проблемами.

Передача именованных аргументов в форме словаря

Как и в случае позиционных аргументов, именованные можно передавать в функцию "пачкой" в виде словаря. Для этого нужно перед словарём поставить две звёздочки.

Рассмотрим определение функции `my_func()`:

```
def my_func(**kwargs):
    print(type(kwargs))
    print(kwargs)
```

Приведенный ниже код:

```
info = {'name':'Timur', 'age':'28', 'job':'teacher'}

my_func(**info)
```

выводит:

```
<class 'dict'>
{'name': 'Timur', 'age': '28', 'job': 'teacher'}
```

Рассмотрим еще один пример определения функции `print_info()`, распечатывающей информацию о пользователе.

```
def print_info(name, surname, age, city, *children, **additional_info):
    print('Имя:', name)
    print('Фамилия:', surname)
    print('Возраст:', age)
    print('Город проживания:', city)
    if len(children) > 0:
        print('Дети:', ', '.join(children))
    if len(additional_info) > 0:
        print(additional_info)
```

Приведенный ниже код:

```
children = ['Бодхи Рансом Грин', 'Ноа Шэннон Грин', 'Джорни Ривер Грин']
additional_info = {'height':163, 'job':'actress'}

print_info('Меган', 'Фокс', 34, 'Ок-Ридж', *children, **additional_info)
```

выводит:

```
Имя: Меган
Фамилия: Фокс
Возраст: 34
Город проживания: Ок-Ридж
Дети: Бодхи Рансом Грин, Ноа Шэннон Грин, Джорни Ривер Грин
{'height': 163, 'job': 'actress'}
```

При подстановке аргументов "разворачивающиеся" наборы аргументов вроде `*positional` и `**named` можно указывать вперемешку с аргументами соответствующего типа: `*positional` с позиционными, а `**named` — с именованными. И, конечно, же, все именованные аргументы должны идти после всех позиционных!

Keyword-only аргументы

В Python 3 добавили возможность пометить именованные аргументы функции так, чтобы вызвать функцию можно было, только передав эти аргументы по именам. Такие аргументы называются `keyword-only` и их нельзя передать в функцию в виде позиционных.

Рассмотрим определение функции `make_circle()`:

```
def make_circle(x, y, radius, *, line_width=1, fill=True):
```

Здесь * выступает разделителем: отделяет обычные аргументы (их можно указывать по имени и позиционно) от строго именованных.

Приведенный ниже код работает как и полагается:

```
make_circle(10, 20, 5)                          # x=10, y=20, radius=5,  line_width=1, fill=True
make_circle(x=10, y=20, radius=7)                          # x=10, y=20, radius=7,  line_width=1, fill=True
make_circle(10, 20, radius=10, line_width=2, fill=False)   # x=10, y=20, radius=10, line_width=2, fill=False
make_circle(x=10, y=20, radius=17, line_width=3)           # x=10, y=20, radius=17, line_width=3, fill=True
```

То есть аргументы `x`, `y` и `radius` могут быть переданы в качестве как позиционных, так и именованных аргументов. При этом аргументы `line_width` и `fill` могут быть переданы **только** как именованные аргументы.

Приведенный ниже код:

```
make_circle(10, 20, 15, 20)
make_circle(x=10, y=20, 15, True)
make_circle(10, 20, 10, 2, False)
```

приводит к возникновению ошибок.

Этот пример неплохо демонстрирует подход к описанию аргументов. Первые три аргумента — координаты центра круга и радиус. Координаты центра и радиус присутствуют у круга всегда, поэтому обязательны и их можно не именовать. А вот `line_width` и `fill` — необязательные аргументы, ещё и получающие ничего не говорящие значения. Вполне логично ожидать, что вызов вида `make_circle(10, 20, 5, 3, False)` мало кому понравится! Ради ясности аргументы `line_width` и `fill` и объявлены так, что могут быть указаны только явно через имя.

Мы также можем объявить функцию, у которой будут только строго именованные аргументы, для этого нужно поставить звёздочку в самом начале перечня аргументов.

```
def make_circle(*, x, y, radius, line_width=1, fill=True):
```

Теперь для вызова функции `make_circle()` нам нужно передать значения всех аргументов явно через их имя:

```
make_circle(x=10, y=20, radius=15)                          # line_width=1, fill=True
make_circle(x=10, y=20, radius=15, line_width=4, fill=False)
```

Такой разделитель можно использовать только один раз в определении функции. Его нельзя применять в функциях с неограниченным количеством позиционных аргументов `*args`.

Примечания.

Примечание 1. Специальный синтаксис `*args` и `**kwargs` в определении функции позволяет передавать функции переменное количество позиционных и именованных аргументов. При этом `args` и `kwargs` **просто имена.** Вы не обязаны их использовать, можно выбрать любые, однако среди Python программистов приняты именно эти.

Примечание 2. Вы можете использовать одновременно `*args` и `**kwargs` в одной строке для вызова функции. В этом случае **порядок имеет значение**. Как и аргументы, не являющиеся аргументами по умолчанию, `*args` должны предшествовать и аргументам по умолчанию, и `**kwargs`. Правильный порядок параметров:

1. позиционные аргументы,
2. `*args` аргументы,
3. `**kwargs` аргументы.

def my_func(a, b, *args, **kwargs):

Парадигмы программирования

1. Парадигмы программирования
2. Императивное программирование
3. Структурное программирование
4. Объектно-ориентированное программирование
5. Логическое программирование
6. Функциональное программирование

Аннотация. Урок посвящен основным парадигмам программирования.

Парадигмы программирования

Парадигма программирования (подход к программированию) — совокупность идей и понятий, определяющих стиль написания компьютерных программ.

Парадигма – устоявшаяся система научных взглядов, в рамках которой ведутся исследования (Т. Кун).

Парадигма программирования определяется:

- вычислительной моделью;
- базовой программной единицей (-ами);
- методами разделения абстракций.

Язык программирования не обязательно использует единственную парадигму. Существуют мультипарадигменные языки. Создатели таких языков считают, что ни одна парадигма не может быть одинаково эффективной для всех задач, и следует позволять программисту выбирать лучшую для решения каждой.

Язык программирования Python – мультипарадигменный.

Основные парадигмы программирования:

- императивное программирование;
- структурное программирование;
- объектно-ориентированное программирование;
- функциональное программирование;
- логическое программирование.

Вычислительная техника создавалась для решения математических задач — расчета баллистических траекторий, численного решения уравнений и т.д. Для этого же предназначены первые языки программирования, такие как Fortran, Алгол, реализованные в парадигме **императивного программирования**.

Императивное программирование

Императивное программирование (ИП) характеризуется тем, что:

- в исходном коде программы записаны инструкции (команды);
- инструкции должны выполняться последовательно;
- данные, полученные при выполнении инструкции, могут записываться в память;

- данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями.

Императивная программа похожа на приказы (англ. imperative — приказ, повелительное наклонение), выражаемые повелительным наклонением в естественных языках. Это последовательность команд, выполняемых процессором.

При императивном подходе к составлению кода широко используется присваивание. Наличие операторов присваивания увеличивает сложность модели вычислений и создает условия для специфических ошибок императивных программ.

Основные механизмы управления:

- последовательное исполнение команд;
- использование именованных переменных;
- использование оператора присваивания;
- использование ветвления (оператор `if`);
- использование безусловного перехода (оператор `goto`).

Ключевой идеей императивного программирования является работа с переменными, как с временным хранением данных в оперативной памяти.

Структурное программирование

Структурная парадигма программирования нацелена на сокращение времени разработки и упрощение поддержки программ за счет использования блочных операторов и подпрограмм. Отличительная черта структурных программ - отказ от оператора безусловного перехода (goto), который широко использовался в 1970 годах.

Основные механизмы управления:

- последовательное исполнение команд;
- использование именованных переменных;
- использование оператора присваивания;
- использование ветвления (оператор `if`);
- использование циклов;
- использование подпрограмм (функций).

В структурном программировании программа по возможности разбивается на маленькие подпрограммы (функции) с изолированным контекстом.

Парадигму структурного программирования предложил нидерландский ученый Эдсгер Дейкстра.

Объектно-ориентированное программирование

В объектно-ориентированной парадигме программа разбивается на объекты - структуры данных, состоящие из полей, описывающих состояние, и методов - функций, применяемых к объектам для изменения или запроса их состояния. Объектно-ориентированную парадигму программирования поддерживают:

- Python;
- C#;

- Java;
- C++;
- JavaScript;
- и другие.

Основные механизмы управления:

- абстракция;
- класс;
- объект;
- полиморфизм;
- инкапсуляция;
- наследование.

Логическое программирование

При использовании логического программирования программа содержит описание проблемы в терминах фактов и логических формул, а решение проблемы система находит с помощью механизмов логического вывода.

В конце 60-х годов XX века Корделл Грин предложил использовать резолюцию как основу логического программирования. Алан Колмеро создал язык логического программирования Prolog в 1971 году. Логическое программирование пережило пик популярности в середине 80-х годов XX века, когда было положено в основу проекта разработки программного и аппаратного обеспечения вычислительных систем пятого поколения.

Важное его преимущество — достаточно высокий уровень машинной независимости, а также возможность откатов, возвращения к предыдущей подцели при отрицательном результате анализа одного из вариантов в процессе поиска решения.

Один из концептуальных недостатков логического подхода — специфичность класса решаемых задач.

Недостаток практического характера — сложность эффективной реализации для принятия решений в реальном времени, скажем, для систем жизнеобеспечения.

Функциональное программирование

Основной инструмент **функционального программирования** (ФП) — математические функции.

Математические функции выражают связь между исходными данными и итогом процесса. Процесс вычисления также имеет вход и выход, поэтому функция — вполне подходящее и адекватное средство описания вычислений. Именно этот простой принцип положен в основу функциональной парадигмы программирования.

Функциональное программирование (ФП) — декларативная парадигма программирования.

Функциональная программа — набор определений функций. Функции определяются через другие функции или рекурсивно через самих себя. При выполнении программы функции получают аргументы, вычисляют и возвращают результат, при необходимости вычисляя значения других функций.

Как преимущества, так и недостатки данной парадигмы определяет модель вычислений без состояний. Если императивная программа на любом этапе исполнения имеет состояние, то есть совокупность значений всех переменных, и производит побочные эффекты, то чисто функциональная программа ни целиком, ни частями состояния не имеет и побочных эффектов не производит. То, что в императивных языках делается путем присваивания значений переменным, в функциональных достигается передачей выражений в параметры функций. В результате чисто функциональная программа не может изменять имеющиеся данные, а может лишь порождать новые копированием и/или расширением старых. Следствие того же — отказ от циклов в пользу рекурсии.

Сильные стороны функционального программирования:

- повышение надёжности кода;
- удобство организации модульного тестирования;
- возможности оптимизации при компиляции;
- возможности параллелизма.

Недостатки: отсутствие присваиваний и замена их на порождение новых данных приводят к необходимости постоянного выделения и автоматического освобождения памяти, поэтому в системе исполнения функциональной программы обязательным компонентом становится высокоэффективный сборщик мусора.

Основные идеи функционального программирования:

- **неизменяемые переменные** — в функциональном программировании можно определить переменную, но изменить ее значение нельзя;
- **чистая функция** — это функция, результат работы которой предсказуем. При вызове с одними и теми же аргументами, такая функция всегда вернет одно и то же значение. Про такие функции говорят, что они не вызывают побочных эффектов;
- **функции высшего порядка** — могут принимать другие функции в качестве аргумента или возвращать их;
- **рекурсия** — поддерживается многими языками программирования, а для функционального программирования обязательна. Дело в том, что в языках ФП отсутствуют циклы, поэтому для повторения операций служит рекурсия. Использование рекурсии в языках ФП оптимизировано, и происходит быстрее, чем в языках императивного программирования;
- **лямбда-выражения** — способ определения анонимных функциональных объектов.

Примечания

Примечание 1. Термин «парадигма программирования» впервые применил в 1978 году Роберт Флойд. В своей лекции при получении премии Тьюринга он отметил, что в программировании можно наблюдать явление, подобное парадигмам Куна, но, в отличие от них, парадигмы программирования не взаимоисключающие: если прогресс искусства программирования в целом требует постоянного изобретения и усовершенствования парадигм, то совершенствование искусства программиста требует расширения репертуара парадигм.

Таким образом, по мнению Роберта Флойда, в отличие от парадигм в научном мире, описанных Куном, парадигмы программирования могут сочетаться, обогащая инструментарий программиста.

Примечание 2. В основе императивных, структурных, объектно-ориентированных языков программирования лежит машина Тьюринга, разработанная Аланом Тьюрингом.

Примечание 3. В основе функциональных языков программирования лежит модель лямбда-исчислений, разработанная Алонзо Чёрчем.

Функции как объекты

Функции как объекты

До сих пор мы рассматривали функции как совершенно отдельный элемент языка со своим синтаксисом и механизмом работы. Но, оказывается, функции также что-то вроде особого типа объектов. Бывают числа, строки, списки, кортежи, словари, множества. А бывают — функции. У каждого из этих типов есть свои операции, свой синтаксис, но все они — объекты.

Напомним, что язык Python мультипарадигменный, он одинаково хорошо поддерживает и функциональную и объектно-ориентированную парадигмы программирования.

Приведенный ниже код:

```
num = 17
numbers = [1, 2, 3]
colors = (1, 2, 3)
name = 'Python'

print(type(num))
print(type(numbers))
print(type(colors))
print(type(name))
```

выводит:

```
<class 'int'>
<class 'list'>
<class 'tuple'>
<class 'str'>
```

Таким образом, целое число — объект типа `int`, список — объект типа `list`, кортеж — объект типа `tuple`, строка — объект типа `str` и т.д.

Любая функция в языке Python — объект типа `function`.

Приведенный ниже код:

```
print(type(print))
print(type(sum))
print(type(abs))
```

выводит:

```
<class 'builtin_function_or_method'>
<class 'builtin_function_or_method'>
<class 'builtin_function_or_method'>
```

Python выдает нам строку `<class 'builtin_function_or_method'>`, которая поясняет, что `print, sum, abs` — встроенные функции языка, или методы.

Обратите внимание: скобки не ставим при передаче аргумента в функцию `type()`, мы не вызываем функцию, а передаем ее название.

Мы также можем объявить свою функцию, которая тоже является объектом.

Приведенный ниже код:

```
def hello():
    print('Hello from function')

print(type(hello))
```

выводит:

```
<class 'function'>
```

В языке Python все объект: число, строка, список, кортеж, множество, словарь, даже функция.

Поскольку функции тоже объекты, работать с ними можно и как с объектами: записывать их в переменные, передавать в качестве аргументов другим функциям, возвращать из функций и т.д.

Приведенный ниже код:

```
def hello():
    print('Hello from function')


func = hello     #  присваиваем переменной func функцию hello
func()           #  вызываем функцию
```

выводит:

```
Hello from function
```

Таким образом, теперь переменную `func` можно использовать как функцию `hello()`.

Рассмотрим еще один пример. Если по какой-либо причине вам не нравится название встроенной функции `print()`, которая выводит указанный текст на экран, можно написать такой код:

```
writeln = print          # как в языке Pascal

writeln('Hello world!')
writeln('Python')
```

Такой код выводит:

```
Hello world!
Python
```

Возможность записать функцию в переменную позволяет гибко управлять тем, какую функциональность мы хотим использовать. В одну и ту же переменную можно записать разные варианты поведения и менять их при необходимости. При этом не только не нужно

будет менять код по всей программе, но и не придется даже изменять код функций. Достаточно переменной присвоить вместо одной функции — другую.

Представим себе ситуацию, когда необходимо выполнить некую функцию, если задано имя команды. Для простоты предположим, если пришла команда `start` — надо выполнить функцию `start()`, если команда `stop` — функцию `stop()`, если команда `pause` — функцию `pause()`.

Такую логику легко можно написать при помощи условного оператора `if .. elif`.

```
def start():
    # тело функции start
    pass

def stop():
    # тело функции stop
    pass

def pause():
    # тело функции pause
    pass

command = input()  # считываем название команды

if command == 'start':
    start()
elif command == 'stop':
    stop()
elif command == 'pause'
    pause()
```

Однако если команд будет много или если их количество будет увеличиваться, то оператор `if` получится слишком громоздким. В этом случае можно создать словарь, где ключом служит название команды, а значением — соответствующая функция.

Приведенный ниже код решает ту же самую задачу, но более гибко:

```
def start():
    # тело функции start
    pass

def stop():
    # тело функции stop
    pass

def pause():
    # тело функции pause
    pass

commands = {'start': start, 'stop': stop, 'pause': pause}  # словарь соответствия команда → функция

command = input()  # считываем название команды
```

```
commands[command]()  # вызываем нужную функцию через словарь по ключу
```

Таким образом, вся логика обработчика команды сводится к очень простой строке:

```
commands[command]()
```

где переменная `command` — имя команды, которую надо выполнить. Такой код намного гибче!

Функции в качестве аргументов других функций

Возможность присваивать имя функции переменной позволяет, в частности, передавать имя функции аргументом другой функции. Это доступно во многих языках, но в Python проще, благодаря его гибкой типизации.

Это позволяет легко и красиво решать сложные задачи. Классическая иллюстрация – математические задачи, где функциональная зависимость играет роль внешнего фактора.

Например, есть программная функция построения графика для заданной математической функции. Если нужно строить графики многих математических функций, то каждый раз придется писать новую функцию, или модифицировать имеющуюся.

Но логика построения графика функции практически не зависит от типа математической функции, поэтому можно рассматривать математическую функцию как аргумент программной функции построения графика. Определим функцию `plot()`, которая принимает 3 аргумента: `f` – функцию, для которой хотим построить график, и `a`, `b` – границы диапазона построения графика.

```
def plot(f, a, b):
```

При запуске функции `plot()` мы можем указать, для какой именно функции строим график. Например, пусть у нас есть следующие математические функции:

```
def square_add_one(x):
    return x*x + 1


def cube_add_square(x):
    return x**3 + x**2
```

Чтобы нарисовать график функции y=x2+1 в диапазоне [1;10], достаточно выполнить функцию `plot()` со следующими аргументами: `plot(square_add_one, 1, 10)`.

Аналогично, вызов функции `plot(cube_add_square, -10, 10)` построит график функции y=x3+x2 в диапазоне [−10;10].

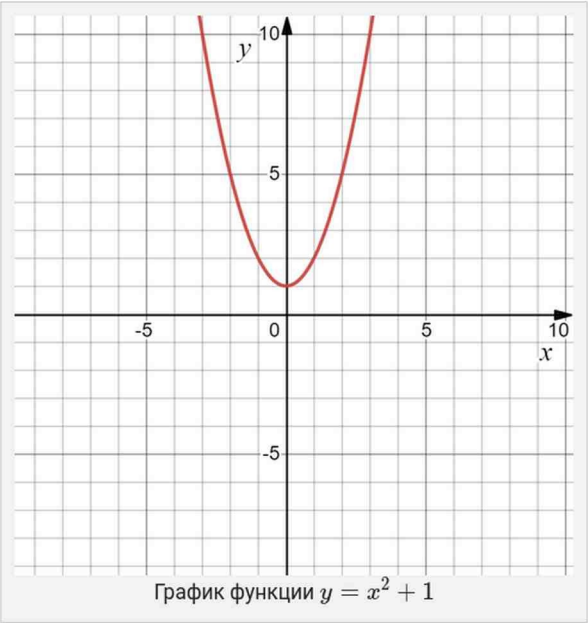


График функции $y = x^2 + 1$

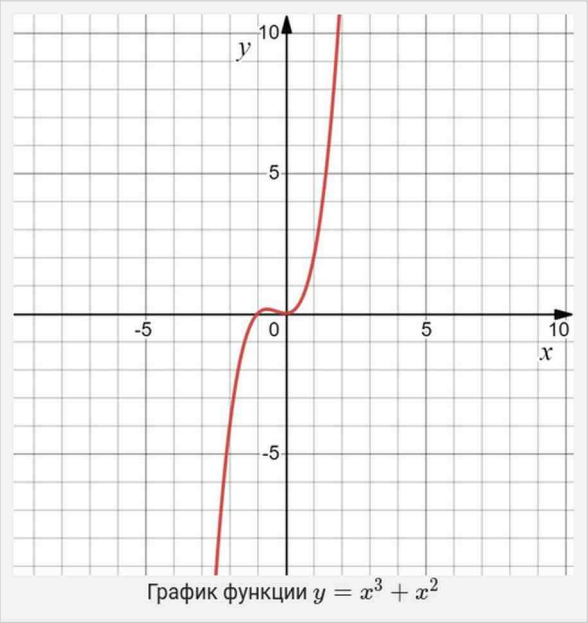


График функции $y = x^3 + x^2$

Логику работы функции `plot()` не нужно менять, достаточно передать ей функциональный объект, используемый внутри. Такой подход работает во многих популярных библиотеках Python. В библиотечные функции мы передаем ту логику, которая нужна нам, вместо стандартной, реализованной в библиотеке.

Функции, способные в качестве аргумента принимать или/и возвращать другие функции, называются функциями **высшего порядка**.

Встроенные функции, принимающие функции в качестве аргументов

В прошлых уроках мы изучили встроенные функции:

- `min()`: поиск минимального элемента;
- `max()`: поиск максимального элемента;
- `sorted()`: сортировка данных.

Не путайте списочный метод `sort()` и встроенную функцию `sorted()`. Они работают одинаково, но списочный метод `sort()` сортирует список на месте, а функция `sorted()` возвращает новый, отсортированный список.

Продемонстрируем работу функций на примере:

```
numbers = [10, -7, 8, -100, -50, 32, 87, 117, -210]

print(max(numbers))
print(min(numbers))
print(sorted(numbers))
```

Приведенный выше код выводит:

```
117
-210
[-210, -100, -50, -7, 8, 10, 32, 87, 117]
```

Но что, если мы хотим написать код для поиска **максимального по модулю** элемента списка `numbers`? И вообще, сравнивать элементы не стандартным способом, а более специфическими?

На этот случай все выше перечисленные встроенные функции могут принимать необязательный аргумент `key` – функцию, определяющую условия сравнения элементов. Другими словами, значение `key` должно быть функцией, принимающей один аргумент и возвращающей на его основе ключ для сравнения.

Функция, определяющая условия сравнения элементов, называется **компаратор** (compare – сравнивать).

Встроенные функции `min()`, `max()`, `sorted()` – функции высшего порядка, так как принимают в качестве аргумента функцию сравнения элементов.

Продемонстрируем вышесказанное на примере кода:

```
numbers = [10, -7, 8, -100, -50, 32, 87, 117, -210]

print(max(numbers, key=abs))       #  указываем функцию abs в качестве компаратора
print(min(numbers, key=abs))       #  указываем функцию abs в качестве компаратора
print(sorted(numbers, key=abs))    #  указываем функцию abs в качестве компаратора
```

Приведенный выше код выводит:

```
-210                                  # максимальный по модулю элемент
-7                                    # минимальный по модулю элемент
[-7, 8, 10, 32, -50, 87, -100, 117, -210]    # сортировка на основании модулей элементов
```

Рассмотрим еще один пример.

Пусть в списке `points` хранятся в виде кортежей координаты точек плоскости в двумерной биполярной системе координат.

```
points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]
```

При использовании встроенной функции `sorted()` (или списочного метода `sort()`) сортировка пройдет по первым значениям пар кортежа, а в случае их совпадения – по вторым.

Таким образом, приведенный ниже код:

```
points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]
points.sort()    #  сортируем список точек на месте
print(points)
```

выводит:

```
[(-10, 15), (1, -1), (1, 5), (2, -4), (2, 3), (7, 18), (10, 9)]
```

Рассмотрим следующий код:

```
def compare_by_second(point):
    return point[1]
```

```
def compare_by_sum(point):
    return point[0] + point[1]


points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

print(sorted(points, key=compare_by_second))   # сортируем по второму значению кортежа
print(sorted(points, key=compare_by_sum))      # сортируем по сумме кортежа
```

Он выводит:

```
[(2, -4), (1, -1), (2, 3), (1, 5), (10, 9), (-10, 15), (7, 18)]
[(2, -4), (1, -1), (2, 3), (-10, 15), (1, 5), (10, 9), (7, 18)]
```

Функции в качестве возвращаемых значений других функций

Объектная сущность функций позволяет и передавать их в качестве аргументов в другие функции, и возвращать одни функции из других. То есть, функции могут быть результатом работы других функций, что позволяет писать генераторы функций, возвращающие функции в зависимости от передаваемых им аргументов.

Рассмотрим код, где функция `generator()` возвращает функцию `hello()` в качестве результата своей работы.

```
def generator():
    def hello():
        print('Hello from function!')
    return hello
```

Результат работы функции `generator()` можно записать в переменную, и использовать эту переменную как функцию.

Приведенный ниже код:

```
func = generator()
func()
```

выводит:

Hello from function!

В Python можно определять функцию внутри функции, ведь функция это объект.

Приведенный выше пример не очень информативен, но идею можно использовать и для построения более мощных генераторов функций. Например, рассмотрим семейство функций — квадратных трехчленов. Все эти функции имеют один и тот же вид f(x)=ax2+bx+c, но поведение конкретного квадратного трехчлена зависит от значения параметров a,b,c. Мы можем написать генератор функций, который по параметрам a,b,c, построит и вернет нам конкретный квадратный трехчлен:

```
def generator_square_polynom(a, b, c):
    def square_polynom(x):
```

```
        return a * x**2 + b * x + c

    return square_polynom
```

Приведенный ниже код:

```
f = generator_square_polynom(a=1, b=2, c=1)
g = generator_square_polynom(a=2, b=0, c=-3)
h = generator_square_polynom(a=-3, b=-10, c=50)

print(f(1))
print(g(2))
print(h(-1))
```

выводит:

```
4
5
57
```

Другими словами мы построили функции $f(x)=x2+2x+1, g(x)=2x2-3, h(x)=-3x2-10x+50$.

Обратите внимание на то, что внутренняя функция `square_polynom()` использует параметры внешней функции `generator_square_polynom()`. Такую вложенную функцию называют **замыканием**.

Замыкания – вложенные функции, ссылающиеся на переменные, объявленные вне определения этой функции, и не являющиеся её параметрами.

Примечания

Примечание 1. Функция `sorted()` и списочный метод `sort()` помимо необязательного аргумента `key` принимают еще аргумент `reverse`, который по умолчанию имеет значение `False`, что соответствует сортировке по возрастанию. Если значение `reverse` установить в `True`, произойдет сортировка по убыванию.

Примечание 2. Сортировка при помощи функции `sorted()` и списочного метода `sort()` стабильна, то есть гарантирует неизменность взаиморасположения равных между собой элементов.

Приведенный ниже код:

```
def comparator(item):
    return item[0]

data = [('red', 1), ('blue', 2), ('green', 5), ('blue', 1)]
data.sort(key=comparator)   # сортируем по первому полю

print(data)
```

выводит:

```
[('blue', 2), ('blue', 1), ('green', 5), ('red', 1)]
```

Обратите внимание, что две записи с `'blue'` сохранили начальный порядок.

Примечание 3. Функции `max()` и `min()` возвращают **первый** максимальный или минимальный элемент, если таковых несколько.

Примечание 4. Надо четко понимать что сделает код `print(input())` и почему это отличается от `print(input)`. Код `print(input())` запросит у пользователя текст и тут же его напечатает, т.к. функция `input()` будет вызвана. Код `print(input)` выдаст нам текстовое представление этой функции: строку `built-in function input`, которая поясняет, что `input` — встроенная функция языка.

Функции высшего порядка

Функции высшего порядка

Как уже сказано, функции, которые принимают или/и возвращают другие функции, называются **функциями высшего порядка**.

Давайте реализуем простейшую функцию высшего порядка:

```
def high_order_function(func):     # функция высшего порядка, так как принимает функцию
    return func(3)

def double(x):                     # обычная функция = функция первого порядка
    return 2*x

def add_one(x):                    # обычная функция = функция первого порядка
    return x + 1
```

Здесь функция `high_order_function()` принимает другую функцию на входе и возвращает результат её вызова с аргументом, равным 3.

Приведенный ниже код:

```
print(high_order_function(double))
print(high_order_function(add_one))
```

выводит:

```
6
4
```

Функции первого порядка принимают и возвращают "обычные" значения, а не функции. Функции же высшего порядка принимают и/или возвращают другие функции.

Функции высшего порядка для обработки набора данных

Часто функции высшего порядка используются для обработки наборов данных. Рассмотрим три важные функции высшего порядка:

- `map()`;
- `filter()`;
- `reduce()`.

В языке Python эти функции уже реализованы, однако для лучшего понимания их работы мы сначала напишем свои версии этих функций, и уже после этого поговорим о встроенных реализациях.

Функции высшего порядка `map()`, `filter()` и `reduce()` довольно широко распространены в функциональном программировании и часто применяются программистами.

Функция map()

При работе со списками часто требуется применить одно и то же преобразование к каждому элементу. Можно написать цикл, содержащий нужное преобразование.

Например, для преобразования списка чисел в список их квадратов, код может выглядеть так:

```
def f(x):
    return x**2     # тело функции, которая преобразует аргумент x

old_list = [1, 2, 4, 9, 10, 25]
new_list = []
for item in old_list:
    new_item = f(item)
    new_list.append(new_item)

print(old_list)
print(new_list)
```

Результатом работы такого кода будет:

```
[1, 2, 4, 9, 10, 25]
[1, 4, 16, 81, 100, 625]
```

Несложно понять, что цикл будет выглядеть одинаково практически во всех случаях. Меняться будет только преобразование, то есть применяемая функция `f()`. Так почему бы не обобщить код, чтобы функция была параметром? Так и сделаем:

```
def map(function, items):
    result = []
    for item in items:
        new_item = function(item)
        result.append(new_item)

    return result
```

Теперь мы можем совершать преобразования, используя функцию высшего порядка `map()`.

Приведенный ниже код:

```
def square(x):
    return x**2

def cube(x):
    return x**3

numbers = [1, 2, -3, 4, -5, 6, -9, 0]

strings = map(str, numbers)        # используем в качестве преобразователя - функцию str
abs_numbers = map(abs, numbers)    # используем в качестве преобразователя - функцию abs

squares = map(square, numbers)     # используем в качестве преобразователя - функцию square
cubes = map(cube, numbers)         # используем в качестве преобразователя - функцию cube
```

```
print(strings)
print(abs_numbers)
print(squares)
print(cubes)
```

выводит:

```
['1', '2', '-3', '4', '-5', '6', '-9', '0']
[1, 2, 3, 4, 5, 6, 9, 0]
[1, 4, 9, 16, 25, 36, 81, 0]
[1, 8, -27, 64, -125, 216, -729, 0]
```

Функция называется "map" то есть "отобразить". Название пришло из математики, где так называется функция, отображающая одно множество значений в другое путём преобразования всех элементов с помощью некой трансформации.

Реализованную нами функцию `map()` можно использовать как альтернативную возможность для преобразования типов элементов любого списка. Раньше мы решали такую задачу с помощью списочных выражений. Теперь можем использовать и функцию `map()`.

Приведенный ниже код при условии, что функция `map()` определена, как указано выше:

```
strings = ['10', '12', '-4', '-9', '0', '1', '23', '100', '99']

numbers1 = [int(c) for c in strings]  # используем списочное выражение для преобразования
numbers2 = map(int, strings)          # используем функцию map() для преобразования

print(numbers1)
print(numbers2)
```

выводит:

```
[10, 12, -4, -9, 0, 1, 23, 100, 99]
[10, 12, -4, -9, 0, 1, 23, 100, 99]
```

Цепочки преобразований

Мы также можем строить цепочки преобразований, несколько раз вызывая функцию `map()`.

Приведенный ниже код при условии, что функция `map()` определена, как указано выше:

```
numbers = ['-1', '20', '3', '-94', '65', '6', '-970', '8']
new_numbers = map(abs, map(int, numbers))
print(new_numbers)
```

выводит:

```
[1, 20, 3, 94, 65, 6, 970, 8]
```

То есть, сначала мы преобразуем список строк в список чисел с помощью кода `map(int, numbers)`, получая список `[-1, 20, 3, -94, 65, 6, -970, 8]`. Далее с помощью еще одного вызова функции `map()` трансформируем полученный список в список `[1, 20, 3, 94, 65, 6, 970, 8]`.

Функция filter()

Другая популярная задача при работе со списками: отобрать часть элементов списка по определенному критерию. Функция высшего порядка для решения такой задачи называется `filter()`.

Функция-критерий, которая возвращает значение `True` или `False`, называется предикатом.

Реализация такой функции может выглядеть так:

```
def filter(function, items):
    result = []
    for item in items:
        if function(item):
            result.append(item)  # добавляем элемент item если функция function вернула значение
                                                                    True

    return result
```

Наша функция `filter()` применяет предикат `function` к каждому элементу и добавляет в итоговый список только те элементы, для которых предикат вернул `True`.

Например, чтобы из исходного списка чисел получить список с элементами, большими 10, можно написать такой код:

```
def is_greater10(num):  # функция возвращает значение True если число больше 10 и False в
противном случае
    return num > 10

numbers = [12, 2, -30, 48, 51, -60, 19, 10, 13]

large_numbers = filter(is_greater10, numbers)  #  список large_numbers содержит элементы,
                                                                    большие 10

print(large_numbers)
```

При условии, что функция `filter()` определена, как указано выше, этот код выводит:

```
[12, 48, 51, 19, 13]
```

Рассмотрим еще пару примеров применения реализованной функции `filter()`.

Приведенный ниже код при условии, что функция `filter()` определена как указано выше:

```
def is_odd(num):
    return num % 2

def is_word_long(word):
    return len(word) > 6

numbers = list(range(15))
words = ['В', 'новом', 'списке', 'останутся', 'только', 'длинные', 'слова']

odd_numbers = filter(is_odd, numbers)
```

```
large_words = filter(is_word_long, words)

print(odd_numbers)
print(large_words)
```

выводит:

```
[1, 3, 5, 7, 9, 11, 13]
['останутся', 'длинные']
```

Функция reduce()

Реализованные нами функции `map()` и `filter()` работали с отдельными элементами списка независимо. Но встречаются циклы с агрегацией результата — формированием одного результирующего значения при комбинации элементов с использованием аргумента-аккумулятора.

Типичные примеры агрегации — сумма всех элементов списка или их произведение.

Приведенный ниже код:

```
numbers = [1, 2, 3, 4, 5]

total = 0
product = 1

for num in numbers:
    total += num
    product *= num

print(total)
print(product)
```

вычисляет сумму и произведение элементов списка и выводит:

```
15
120
```

С точки зрения математики сумма 1+2+3+4+5 может быть выражена как:

(((((0+1)+2)+3)+4)+5).Нуль здесь – тот самый аккумулятор, точнее его начальное значение. Он не добавляет к сумме ничего, поэтому может служить отправной точкой. А еще нуль будет результатом, если входной список пуст.

Несложно понять, что этот цикл будет выглядеть одинаково практически во всех случаях. Меняться будет только начальное значение аккумулятора (0 для суммы, 1 для произведения и т.д.) и операция, которая комбинирует элемент и аккумулятор. Так почему бы не обобщить этот код? Так и сделаем:

```
def reduce(operation, items, initial_value):
    acc = initial_value
    for item in items:
        acc = operation(acc, item)
```

```
    return acc
```

Приведенный ниже код, при условии, что функция `reduce()` определена как указано выше:

```
def add(x, y):
    return x+y

def mult(x, y):
    return x*y

numbers = [1, 2, 3, 4, 5]

total = reduce(add, numbers, 0)
product = reduce(mult, numbers, 1)

print(total)
print(product)
```

выводит:

```
15
120
```

Примечания

Примечание 1. Мы с вами реализовали три функции:

- `map()`: преобразование элементов списка;
- `filter()`: фильтрация элементов списка;
- `reduce()`: агрегация элементов списка.

Каждая функция имеет меньшую мощность, чем цикл `for`. Цикл `for` позволяет гибко управлять процессом итерации, мы можем использовать даже команды `break` и `continue`. Возникает резонный вопрос: зачем нужны отдельные функции, когда можно обойтись циклом?

Во-первых, такие функции — часть функционального подхода.

Во-вторых, каждая такая функция делает единственную работу, что значительно упрощает рассуждение о коде, его чтение и написание. Взглянув на имя функции, можно понять, что `filter()` отфильтрует, а `map()` преобразует элементы. Более того, по построению функция `filter()` не меняет элементы, а лишь отбрасывает их часть. А функция `map()` меняет значение элементов, но не меняет их количество и позиции.

Примечание 2. В математике определенная нами функция `reduce()` называется левая свёртка (left fold), по сути, мы сворачиваем список в одно значение, начиная слева. Существует ещё и правая свёртка (right fold). В большинстве случаев обе свёртки дают одинаковый результат, если применяемая операция ассоциативна.

Встроенные функции map(), filter(), reduce()

Встроенные функции map(), filter(), reduce()

Язык Python имеет встроенные реализации функций высшего порядка `map()`, `filter()` и `reduce()` , которые намного удобнее, чем наши собственные версии.

Встроенная функция map()

Встроенная функция `map()` имеет сигнатуру `map(func, *iterables)`. В отличие от нашей версии из прошлого урока, встроенная функция `map()` может принимать сразу несколько последовательностей, переменное количество аргументов.

В качестве параметра `func` указывается функция, которой будет передаваться текущий элемент последовательности. Внутри функции `func` необходимо вернуть новое значение. Для примера прибавим к каждому элементу списка число 7.

Приведенный ниже код:

```
def increase(num):
    return num + 7

numbers = [1, 2, 3, 4, 5, 6]
new_numbers = map(increase, numbers)     #  используем встроенную функцию map()

print(new_numbers)
```

выведет не список, а специальный объект:

```
<map object at 0x...>
```

Такой объект похож на список тем, что его можно перебирать циклом `for`, то есть итерировать. Такие объекты в Python называют **итераторами**.

Приведенный ниже код:

```
def increase(num):
    return num + 7

numbers = [1, 2, 3, 4, 5, 6]
new_numbers = map(increase, numbers)

for num in new_numbers:    #  итерируем циклом for
    print(num)
```

выводит:

```
8
9
```

```
10
11
12
13
```

Чтобы получить из итератора список, нужно воспользоваться функцией `list()`:

```
new_numbers = list(map(increase, numbers))
```

Функция `map()` возвращает объект, поддерживающий итерации, а не список. Чтобы получить из него список, необходимо результат передать в функцию `list()`.

Функции `map()` можно передать несколько последовательностей. В этом случае в функцию обратного вызова `func` будут передаваться сразу несколько элементов, расположенных в последовательностях на одинаковых позициях.

Приведенный ниже код суммирует элементы трех списков:

```
def func(elem1, elem2, elem3):
    return elem1 + elem2 + elem3

numbers1 = [1, 2, 3, 4, 5]
numbers2 = [10, 20, 30, 40, 50]
numbers3 = [100, 200, 300, 400, 500]

new_numbers = list(map(func, numbers1, numbers2, numbers3))  # преобразуем итератор в список

print(new_numbers)
```

и выводит:

```
[111, 222, 333, 444, 555]
```

Если в последовательностях разное количество элементов, то последовательность с минимальным количеством элементов становится ограничителем.

Приведенный ниже код:

```
def func(elem1, elem2, elem3):
    return elem1 + elem2 + elem3

numbers1 = [1, 2, 3, 4]
numbers2 = [10, 20]
numbers3 = [100, 200, 300, 400, 500]

new_numbers = list(map(func, numbers1, numbers2, numbers3))  # преобразуем итератор в список

print(new_numbers)
```

выводит:

```
[111, 222]
```

Встроенная функция `map()` реализована очень гибко. В качестве последовательностей мы можем использовать: списки, строки, кортежи, множества, словари.

Приведем пример удобного использования встроенной функции `map()`, которой передано две последовательности.

Приведенный ниже код:

```
circle_areas = [3.56773, 5.57668, 4.31914, 6.20241, 91.01344, 32.01213]

result1 = list(map(round, circle_areas, [1]*6))        # округляем числа до 1 знака после запятой
result2 = list(map(round, circle_areas, range(1, 7)))   # округляем числа до 1,2,...,6 знаков после запятой

print(circle_areas)
print(result1)
print(result2)
```

выводит:

```
[3.56773, 5.57668, 4.31914, 6.20241, 91.01344, 32.01213]
[3.6, 5.6, 4.3, 6.2, 91.0, 32.0]
[3.6, 5.58, 4.319, 6.2024, 91.01344, 32.01213]
```

Встроенная функция `round(x, n=0)` принимает два числовых аргумента `x` и `n` и округляет переданное число `x` до `n` цифр после десятичной запятой. Значением по умолчанию для `n` является 0.

Встроенная функция filter()

Встроенная функция `filter()` имеет сигнатуру `filter(func, iterable)`. В отличие от нашей реализации из прошлого урока она может принимать любой итерируемый объект (список, строку, кортеж, и т.д.).

В качестве параметра `func` указывается ссылка на функцию, которой будет передаваться текущий элемент последовательности. Внутри функции `func` необходимо вернуть значение `True` или `False`. Для примера, удалим все отрицательные значения из списка.

Приведенный ниже код:

```
def func(elem):
    return elem >= 0

numbers = [-1, 2, -3, 4, 0, -20, 10]
positive_numbers = list(filter(func, numbers))  #  преобразуем итератор в список

print(positive_numbers)
```

выводит:

```
[2, 4, 0, 10]
```

Обратите внимание: функция `filter()` как и функция `map()` возвращает не список, а специальный объект, который называется итератором. Итераторы можно перебрать с помощью цикла `for`, либо преобразовать в список.

Встроенной функции `filter()` можно в качестве первого параметра `func` передать значение `None`. В таком случае каждый элемент последовательности будет проверен на соответствие значению `True`. Если элемент в логическом контексте возвращает значение `False`, то он не будет добавлен в возвращаемый результат.

О преобразовании основных типов в булево значение (`True/False`) можно почитать в этом уроке.

Приведенный ниже код:

```
true_values = filter(None, [1, 0, 10, '', None, [], [1, 2, 3], ()])

for value in true_values:
    print(value)
```

выводит:

```
1
10
[1, 2, 3]
```

В данном случае, значения списка: `0, '', None, [], ()` позиционируется как `False`, а значения `1, 10, [1, 2, 3]` как `True`.

Функция reduce()

Для использования функции `reduce()` необходимо подключить специальный модуль `functools`. Функция `reduce()` имеет сигнатуру `reduce(func, iterable, initializer=None)`. Если начальное значение не установлено, то в его качестве используется первое значение из последовательности `iterable`.

Приведенный ниже код:

```
from functools import reduce

def func(a, b):
    return a + b

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
total = reduce(func, numbers, 0)   # в качестве начального значения 0
print(total)
```

выводит:

```
55
```

Обратите внимание на то, что мы могли вызвать функцию так:

```
total = reduce(func, numbers)    # в качестве начального значения первый элемент списка numbers
```

Функция `reduce()` во второй версии языка Python была встроенной, но в Python 3 ее решили перенести в модуль `functools`. Функция `reduce()` как и функции `map()` и `filter()` может принимать любой итерируемый объект.

Модуль operator

Чтобы не писать каждый раз функции, определяющие такие стандартные математические операции как сумма или произведение:

```
def sum_func(a, b):
    return a + b

def mult_func(a, b):
    return a * b
```

можно использовать уже реализованные функции из модуля `operator`.

Неполный список функций из модуля `operator` выглядит так:

Операция	Синтаксис	Функция
Addition	a + b	add(a, b)
Containment Test	obj in seq	contains(seq, obj)
Division	a / b	truediv(a, b)
Division	a // b	floordiv(a, b)
Exponentiation	a ** b	pow(a, b)
Modulo	a % b	mod(a, b)
Multiplication	a * b	mul(a, b)
Negation (Arithmetic)	-a	neg(a)
Subtraction	a - b	sub(a, b)
Ordering	a < b	lt(a, b)
Ordering	a <= b	le(a, b)
Equality	a == b	eq(a, b)
Difference	a != b	ne(a, b)
Ordering	a >= b	ge(a, b)
Ordering	a > b	gt(a, b)

Рассмотрим примеры использования функций из модуля `operator`.

Приведенный ниже код:

```
from operator import *    #  импортируем все функции

print(add(10, 20))        #  сумма
print(floordiv(20, 3))    #  целочисленное деление
print(neg(9))             #  смена знака
print(lt(2, 3))           #  проверка на неравенство <
```

```
print(lt(10, 8))          #  проверка на неравенство <
print(eq(5, 5))           #  проверка на равенство ==
print(eq(5, 9))           #  проверка на равенство ==
```

выводит:

```
30
6
-9
True
False
True
False
```

Приведенный ниже код:

```
from functools import reduce
import operator

words = ['Testing ', 'shows ', 'the ', 'presence', ', ', 'not ', 'the ', 'absence ', 'of ', 'bugs']
numbers = [1, 2, -6, -4, 3, 9, 0, -6, -1]

opposite_numbers = list(map(operator.neg, numbers))    #  смена знаков элементов списка
concat_words = reduce(operator.add, words)             #  конкатенация элементов списка

print(opposite_numbers)
print(concat_words)
```

Выводит:

```
[-1, -2, 6, 4, -3, -9, 0, 6, 1]
Testing shows the presence, not the absence of bugs
```

Модуль `operator` реализован на языке `C`, поэтому функции этого модуля работают в разы быстрее, чем самописные функции в Python.

Примечания

Примечание 1. Итераторы – важная концепция языка Python. Нужно помнить:

- итераторы можно обойти циклом `for`;
- итератор можно преобразовать в список или кортеж, с помощью функций `list()` и `tuple()`;
- итератор можно **распаковать** с помощью *

Приведенный ниже код:

```
numbers = [1, 10, -9, 8, 9, 345, -32, -89, 2]

map_obj = map(abs, numbers)

print(*map_obj)           # распаковываем
```

выводит:

```
1 10 9 8 9 345 32 89 2
```

Примечание 2. Если нам нужны строковые методы в виде функций, мы можем получить их через название типа `str`.

Приведенный ниже код:

```
pets = ['alfred', 'tabitha', 'william', 'arla']
chars = ['x', 'y', '2', '3', 'a']

uppered_pets = list(map(str.upper, pets))
capitalized_pets = list(map(str.capitalize, pets))
only_letters = list(filter(str.isalpha, chars))

print(uppered_pets)
print(capitalized_pets)
print(only_letters)
```

выводит:

```
['ALFRED', 'TABITHA', 'WILLIAM', 'ARLA']
['Alfred', 'Tabitha', 'William', 'Arla']
['x', 'y', 'a']
```

Аналогично можно получить методы других типов данных в виде функций.

Примечание 3. Подробнее прочитать про модуль `operator` можно в официальной документации.

Примечание 4. В уроке, посвященном списочным выражениям, мы разбирали конструкции очень похожие на действие функций `map()` и `filter()`. Списочное выражение можно рассматривать как комбинацию фильтрации и трансформации: сначала выполняется фильтрация, затем — трансформирование.

Анонимные функции

Анонимные функции

Помимо стандартного определения функции, состоящего из ее заголовка с ключевым словом `def` и блока кода – тела функции, в Python можно создавать короткие однострочные функции с использованием оператора `lambda`. Это анонимные функции или лямбда-функции.

Анонимные функции – функции с телом, но без имени.

Общий формат определения анонимной функции: `lambda список_параметров: выражение`.

Тут `список_параметров` – список параметров через запятую, `выражение` – значение, либо код, дающий значение.

Параметры анонимных функций, в отличие от обычных, не нужно заключать в скобки.

Следующие два определения функций эквивалентны:

```
def standard_function(x):           #  стандартное объявление функции
    return x*2

lambda_function = lambda x: x*2      #  объявление анонимной функции
```

Приведенный ниже код:

```
print(standard_function(7))
print(lambda_function(7))
```

выводит:

```
14
14
```

Рассмотрим еще примеры. Приведенный ниже код:

```
f1 = lambda: 10 + 20              # функция без параметров
f2 = lambda x, y: x + y           # функция с двумя параметрами
f3 = lambda x, y, z: x + y + z    # функция с тремя параметрами

print(f1())
print(f2(5, 10))
print(f3(5, 10, 30))
```

выводит:

```
30
15
45
```

Когда применение анонимных функций оправдано:

- однократное использование функции;
- передача функций в качестве аргументов другим функциям;
- возвращение функции в качестве результата другой функции.

Однократное использование функции

В одном из прошлых уроков нам приходилось сортировать список кортежей. Как мы уже знаем, встроенная функция `sorted()` (или списочный метод `sort()`) сортируют по первым значениям кортежей, а в случае их совпадения, по вторым и т.д. Для сортировки, отличной от стандартной нам приходилось создавать отдельные функции-компараторы для сравнения элементов:

```
def compare_by_second(point):
    return point[1]

def compare_by_sum(point):
    return point[0] + point[1]

points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

print(sorted(points, key=compare_by_second))   # сортируем по второму значению кортежа
print(sorted(points, key=compare_by_sum))      # сортируем по сумме кортежа
```

Очевидно, что такие функции как `compare_by_second()` и `compare_by_sum()` не особо нужны вне контекста сортировки, поэтому логично их заменить на анонимные функции:

```
points = [(1, -1), (2, 3), (-10, 15), (10, 9), (7, 18), (1, 5), (2, -4)]

print(sorted(points, key=lambda point: point[1]))                # сортируем по второму значению кортежа
print(sorted(points, key=lambda point: point[0] + point[1]))     # сортируем по сумме элементов кортежа
```

Название аргумента `point` в анонимной функции можно заменить на любое другое.

Передача анонимных функций в качестве аргументов другим функциям

Функции высшего порядка `map()`, `filter()` и `reduce()` идеально подойдут для демонстрации удобства анонимных функций в качестве аргументов других функций.

Теперь нет необходимости делать преобразующую элементы функцию отдельно определенной именованной функцией.

Приведенный ниже код:

```
numbers = [1, 2, 3, 4, 5, 6]

new_numbers1 = list(map(lambda x: x+1, numbers))     #  увеличиваем на 1
```

```
new_numbers2 = list(map(lambda x: x*2, numbers))      #  удваиваем
new_numbers3 = list(map(lambda x: x**2, numbers))     #  возводим в квадрат

print(new_numbers1)
print(new_numbers2)
print(new_numbers3)
```

выводит:

```
[2, 3, 4, 5, 6, 7]
[2, 4, 6, 8, 10, 12]
[1, 4, 9, 16, 25, 36]
```

Приведенный ниже код:

```
strings = ['a', 'b', 'c', 'd', 'e']
numbers = [3, 2, 1, 4, 5]

new_strings = list(map(lambda x, y: x*y, strings, numbers))

print(new_strings)
```

выводит:

```
['aaa', 'bb', 'c', 'dddd', 'eeeee']
```

Рассмотрим примеры использования анонимных функций в качестве аргумента функции `filter()`.

Приведенный ниже код:

```
numbers = [-1, 2, -3, 4, 0, -20, 10, 30, -40, 50, 100, 90]

positive_numbers = list(filter(lambda x: x > 0, numbers))      #  положительные числа
large_numbers = list(filter(lambda x: x > 50, numbers))        #  числа, большие 50
even_numbers = list(filter(lambda x: x % 2 == 0, numbers))     #  четные числа

print(positive_numbers)
print(large_numbers)
print(even_numbers)
```

выводит:

```
[2, 4, 10, 30, 50, 100, 90]
[100, 90]
[2, 4, 0, -20, 10, 30, -40, 50, 100, 90]
```

Приведенный ниже код:

```
words = ['python', 'stepik', 'beegeek', 'iq-option']

new_words1 = list(filter(lambda w: len(w) > 6, words))    #  слова длиною больше 6 символов
new_words2 = list(filter(lambda w: 'e' in w, words))      #  слова содержащие букву e
```

```
print(new_words1)
print(new_words2)
```

выводит:

```
['beegeek', 'iq-option']
['stepik', 'beegeek']
```

Рассмотрим примеры использования анонимных функций в качестве аргументов функции `reduce()`.

Приведенный ниже код:

```
from functools import reduce

words = ['python', 'stepik', 'beegeek', 'iq-option']
numbers = [1, 2, 3, 4, 5, 6]

summa = reduce(lambda x, y: x + y, numbers, 0)
product = reduce(lambda x, y: x * y, numbers, 1)
sentence = reduce(lambda x, y: x + ' loves ' + y, words, 'Everyone')

print(summa)
print(product)
print(sentence)
```

выводит:

```
21
720
Everyone loves python loves stepik loves beegeek loves iq-option
```

Возвращение функции в качестве результата другой функции

Анонимные функции могут быть результатом работы других функций.

Приведенный ниже код по значениям a,b,c строит и возвращает квадратный трехчлен:

```
def generator_square_polynom(a, b, c):
    def square_polynom(x):
        return a*x**2 + b*x + c
    return square_polynom
```

Такой код можно переписать так:

```
def generator_square_polynom(a, b, c):
    return lambda x: a*x**2 + b*x + c
```

Этот пример показывает, что анонимные функции являются **замыканиями**: возвращаемая функция запоминает значения переменных `a, b, c` из внешнего окружения.

Условный оператор в теле анонимной функции

В теле анонимной функции не получится выполнить несколько действий и не получится использовать многострочные конструкции вроде циклов `for` и `while`. Однако можно использовать тернарный условный оператор.

Приведенный ниже код:

```
numbers = [-2, 0, 1, 2, 17, 4, 5, 6]

result = list(map(lambda x: 'even' if x % 2 == 0 else 'odd', numbers))

print(result)
```

выводит:

```
['even', 'even', 'odd', 'even', 'odd', 'even', 'odd', 'even']
```

Общий вид тернарного условного оператора в теле анонимной функции выглядит так:

значение1 if условие else значение2

Если `условие` истинно, возвращается `значение1` , если нет – `значение2`.

Анонимные функции не могут содержать многострочных конструкций, зато их легко читать. Этого сложно было бы добиться, разреши авторы "многострочные" анонимные функции.

Передача аргументов в анонимную функцию

Как и обычные функции, определенные с помощью ключевого слова `def` , анонимные функции поддерживают все способы передачи аргументов:

- позиционные аргументы;
- именованные аргументы;
- переменный список позиционных аргументов (`*args`);
- переменный список именованных аргументов (`**kwargs`);
- обязательные аргументы (`*`).

Приведенный ниже код:

```
f1 = lambda x, y, z: x + y + z
f2 = lambda x, y, z=3: x + y + z
f3 = lambda *args: sum(args)
f4 = lambda **kwargs: sum(kwargs.values())
f5 = lambda x, *, y=0, z=0: x + y + z


print(f1(1, 2, 3))
print(f2(1, 2))
print(f2(1, y=2))
print(f3(1, 2, 3, 4, 5))
print(f4(one=1, two=2, three=3))
print(f5(1))
```

```
print(f5(1, y=2, z=3))
```

выводит:

```
6
6
6
15
6
1
6
```

Ограничения анонимных функций

Особенности и ограничения анонимных функций в Python:

- анонимная функция может содержать только выражение, и не может включать в свое тело операторы;
- в теле анонимной функции такие операторы, как `return`, `pass`, `assert` или `raise`, вызовут исключение `SyntaxError`;
- анонимная функция пишется как одна строка исполнения;
- анонимная функция может быть немедленно вызвана.

Примечания

Примечание 1. Интересная особенность анонимных функций (лямбда-функций) – они являются выражениями. После определения лямбда-функции ее можно сразу же вызвать.

Приведенный ниже код:

```
print((lambda x, y: x + y)(5, 10))     # 5 + 10
print(1 + (lambda x: x*5)(10) + 2)     # 1 + 50 + 2
```

выводит:

```
15
53
```

Анонимные функции можно конструировать и тут же вызывать. На практике это редко применяется.

Примечание 2. В лямбда исчислении, часто применяемом в разработке языков программирования, все функции — анонимные, поэтому анонимные функции во многих языках тоже называют "лямбдами" или "лямбда-функциями".

Примечание 3. В Python анонимные функции — лишь сокращенная запись функции, поэтому приведенный ниже код:

```
f = lambda x: x + 1
print(type(f))
```

выводит:

```
<class 'function'>
```

То есть, анонимные функции имеют такой же тип, как и обычные функции.

Примечание 4. Анонимные функции **очень часто** используются вместе со встроенными функциями `map()`, `filter()`, `reduce()`, `sorted()`, `max()`, `min()` и т.д.

Встроенные функции any(), all(), zip(), enumerate()

Функции all() и any()

При работе с коллекциями часто приходится определять, выполняется ли некоторое условие одновременно для всех элементов последовательности или хотя бы для одного. Для этого существуют две встроенные функции all() и any().

Функция all()

Встроенная функция all() возвращает значение True, если **все элементы** переданной ей последовательности (итерируемого объекта) истинны (приводятся к значению True), и False в противном случае.

Сигнатура функции следующая: all(iterable). В качестве iterable может выступать любой итерируемый объект:

- список;
- кортеж;
- строка;
- множество;
- словарь и т.д.

Приведенный ниже код:

```
print(all([True, True, True]))    # возвращает True, так как все значения списка равны True
print(all([True, True, False]))   # возвращает False, так как не все значения списка равны True
```

выводит:

```
True
False
```

Напомним, что в Python все следующие значения приводятся к значению False:

- константы None и False;
- нули всех числовых типов данных: 0, 0.0, 0j, Decimal(0), Fraction(0, 1);
- пустые коллекции: '', (), [], {}, set(), range(0).

Приведенный ниже код:

```
print(all([1, 2, 3]))
print(all([1, 2, 3, 0, 5]))
print(all([True, 0, 1]))
print(all(('', 'red', 'green')))
print(all({0j, 3+4j}))
```

выводит:

```
True
False
False
```

```
False
False
```

При работе со словарями функция all() проверяет на соответствие параметрам True ключи словаря, а не их значения.

Приведенный ниже код:

```
dict1 = {0: 'Zero', 1: 'One', 2: 'Two'}
dict2 = {'Zero': 0, 'One': 1, 'Two': 2}

print(all(dict1))
print(all(dict2))
```

выводит:

```
False
True
```

Обратите внимание: если переданный итерируемый объект пустой, то функция all() возвращает значение True.

Приведенный ниже код:

```
print(all([]))          #  передаем пустой список
print(all(()))          #  передаем пустой кортеж
print(all(''))          #  передаем пустую строку
print(all([[], []]))    #  передаем список, содержащий пустые списки
```

выводит:

```
True
True
True
False
```

Функция any()

Встроенная функция any() возвращает значение True, если **хотя бы один элемент** переданной ей последовательности (итерируемого объекта) является истинным (приводится к значению True), и False в противном случае.

Сигнатура функции следующая: any(iterable). В качестве iterable может выступать любой итерируемый объект:

- список;
- кортеж;
- строка;
- множество;
- словарь и т.д.

Приведенный ниже код:

```
print(any([False, True, False]))       #  возвращает True, так как есть хотя бы один элемент, равный True
```

```
print(any([False, False, False]))      #  возвращает False, так как нет элементов, равных True
```

выводит:

```
True
False
```

Приведенный ниже код:

```
print(any([0, 0, 0]))
print(any([0, 1, 0]))
print(any([False, 0, 1]))
print(any(['', [], 'green']))
print(any({0j, 3+4j, 0.0}))
```

выводит:

```
False
True
True
True
True
```

При работе со словарями функция any() проверяет на соответствие True ключи словаря, а не их значения.

Приведенный ниже код:

```
dict1 = {0: 'Zero'}
dict2 = {'Zero': 0, 'One': 1}

print(any(dict1))
print(any(dict2))
```

выводит:

```
False
True
```

Обратите внимание: если переданный объект пуст, то функция any() возвращает значение False.

Приведенный ниже код:

```
print(any([]))          #  передаем пустой список
print(any(()))          #  передаем пустой кортеж
print(any(''))          #  передаем пустую строку
print(any([[], []]))    #  передаем список, содержащий пустые списки
```

выводит:

```
False
False
False
False
```

Функции all() и any() в связке с функцией map()

Функции all() и any() могут быть полезны в комбинации с функцией map(), которая может преобразовывать элементы последовательности (итерируемого объекта) к значению True/False в соответствии с неким условием.

Приведенный ниже код, проверяет, **все ли элементы** списка numbers больше 10:

```
numbers = [17, 90, 78, 56, 231, 45, 5, 89, 91, 11, 19]

result = all(map(lambda x: True if x > 10 else False, numbers))

if result:
    print('Все числа больше 10')
else:
    print('Хотя бы одно число меньше или равно 10')
```

и выводит:

```
Хотя бы одно число меньше или равно 10
```

так как список numbers содержит число 5, которое не больше чем 10.

Лямбда функцию, которая преобразует элементы списка numbers в значения True/False можно упростить следующим образом:

```
result = all(map(lambda x: x > 10, numbers))
```

Приведенный ниже код, проверяет, что **хотя бы один элемент** списка четное число:

```
numbers = [17, 91, 78, 55, 231, 45, 5, 89, 99, 11, 19]

result = any(map(lambda x: x % 2 == 0, numbers))

if result:
    print('Хотя бы одно число четное')
else:
    print('Все числа нечетные')
```

и выводит:

```
Хотя бы одно число четное
```

так как список numbers содержит четное число 78.

Функция enumerate()

Встроенная функция enumerate() возвращает кортеж из индекса элемента и самого элемента переданной ей последовательности (итерируемого объекта).

Сигнатура функции следующая: enumerate(iterable, start). В качестве iterable может выступать любой итерируемый объект:

- список;
- кортеж;

- строка;
- множество;
- словарь и т.д.

С помощью необязательного параметра start можно задать начальное значение индекса. По умолчанию значение параметра start = 0, то есть счет начинается с нуля.

Приведенный ниже код:

```
colors = ['red', 'green', 'blue']

for pair in enumerate(colors):
    print(pair)
```

выводит:

```
(0, 'red')
(1, 'green')
(2, 'blue')
```

Если счет нужно начать с отличного от нуля числа, то нужно передать значение аргумента start.

Приведенный ниже код:

```
colors = ['red', 'green', 'blue']

for pair in enumerate(colors, 100):
    print(pair)
```

выводит:

```
(100, 'red')
(101, 'green')
(102, 'blue')
```

Обратите внимание, функция enumerate() возвращает не список, а специальный объект, который называется итератором. Такой объект похож на список тем, что его можно перебирать циклом for, то есть итерировать. Итератор можно преобразовать в список с помощью функции list().

Приведенный ниже код:

```
colors = ['red', 'green', 'blue']

pairs = enumerate(colors)

print(pairs)
print(list(pairs))
```

выводит:

```
<enumerate object at 0x...>
[(0, 'red'), (1, 'green'), (2, 'blue')]
```

Мы также можем использовать распаковку кортежей при итерировании с помощью цикла for.

Приведенный ниже код:

```
colors = ['red', 'green', 'blue']
for index, item in enumerate(colors):
    print(index, item)
```

выводит:

```
0 red
1 green
2 blue
```

Такой код является альтернативой коду:

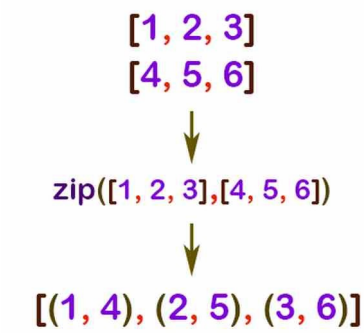
```
colors = ['red', 'green', 'blue']
for i in range(len(colors)):
    print(i, colors[i])
```

Функция zip()

Встроенная функция zip() объединяет отдельные элементы из каждой переданной ей последовательности (итерируемого объекта) в кортежи.

Сигнатура функции следующая: zip(*iterables). В качестве iterable может выступать любой итерируемый объект:

- список;
- кортеж;
- строка;
- множество;
- словарь и т.д.



Приведенный ниже код:

```
numbers = [1, 2, 3]
words = ['one', 'two', 'three']

for pair in zip(numbers, words):
    print(pair)
```

выводит:

```
(1, 'one')
(2, 'two')
(3, 'three')
```

Функция zip(), как и функция enumerate() возвращает не список, а специальный объект, который называется итератором. Такой объект похож на список тем, что его можно перебирать циклом for, то есть итерировать. Итератор можно преобразовать в список с помощью функции list().

Приведенный ниже код:

```
numbers = [1, 2, 3]
words = ['one', 'two', 'three']

result = zip(numbers, words)

print(result)
print(list(result))
```

выводит:

```
<zip object at 0x...>
[(1, 'one'), (2, 'two'), (3, 'three')]
```

Мы можем передавать функции zip() сколько угодно итерируемых объектов.

Приведенный ниже код:

```
numbers = [1, 2, 3]
words = ['one', 'two', 'three']
romans = ['I', 'II', 'III']

result = zip(numbers, words, romans)
print(list(result))
```

выводит:

```
[(1, 'one', 'I'), (2, 'two', 'II'), (3, 'three', 'III')]
```

Мы можем передать функции zip() даже один итерируемый объект.

Приведенный ниже код:

```
numbers = [1, 2, 3]
result = zip(numbers)
print(list(result))
```

выводит:

```
[(1,), (2,), (3,)]
```

Если функции zip() передать итерируемые объекты, имеющие разную длину, то объект с наименьшим количеством элементов определяет итоговую длину.

Приведенный ниже код:

```
numbers = [1, 2, 3, 4]
words = ['one', 'two']
romans = ['I', 'II', 'III']

result = zip(numbers, words, romans)
print(list(result))
```

выводит:

```
[(1, 'one', 'I'), (2, 'two', 'II')]
```

Частые сценарии использования функции zip()

Сценарий 1. Функция zip() удобна для создания словарей, когда ключи и значения находятся в разных списках.

Приведенный ниже код:

```
keys = ['name', 'age', 'gender']
values = ['Timur', 28, 'male']

info = dict(zip(keys, values))
print(info)
```

выводит:

```
{'name': 'Timur', 'age': 28, 'gender': 'male'}
```

Сценарий 2. Функция zip() удобна для одновременного (параллельного) итерирования сразу по нескольким коллекциям.

Приведенный ниже код:

```
name = ['Timur', 'Ruslan', 'Rustam']
age = [28, 21, 19]

for x, y in zip(name, age):
    print(x, y)
```

выводит:

```
Timur 28
Ruslan 21
Rustam 19
```

Примечания

Примечание 1. Итераторы – важная концепция языка Python. Нужно помнить:

- итераторы можно обойти циклом for;
- итератор можно преобразовать в список или кортеж, с помощью функций list() и tuple();
- итератор можно распаковать с помощью *.

Примечание 2. Реализация встроенных функций all() и any() выглядит *примерно* так:

```
def all(iterable):
    for item in iterable:
        if not item:
            return False
    return True

def any(iterable):
    for item in iterable:
        if item:
            return True
    return False
```

Примечание 3. Мы можем использовать одновременно функции zip() и enumerate():

Приведенный ниже код:

```
list1 = ['a1', 'a2', 'a3']
list2 = ['b1', 'b2', 'b3']

for index, (item1, item2) in enumerate(zip(list1, list2)):
    print(index, item1, item2)
```

выводит:

```
0 a1 b1
1 a2 b2
2 a3 b3
```