KARPOV.COURSES >>>
КОНСПЕКТ

> Конспект > 2 урок > Функции. Ссылочная модель данных. Погружение в типы. Изменяемые типы. Срезы. Работа со строками

> Оглавление

> Функции в python

Функция – это мини-программа внутри вашей основной программы, которая делает какую-то одну понятную вещь. Вы однажды описываете, что это за вещь, а потом ссылаетесь на это описание.

Например, вы написали кусочек кода, который выполняет определенную задачу и вам ее дальше нужно использовать несколько раз. Вы можете просто объявить функцию, написать этот код, а дальше за место этого полотна кода писать только название этой функции и вызывать его.

Т.е. функции нужны для переиспользования кода

Сохранение действий в одно имя помогает разработке:

1. Уменьшается количество строк кода
2. Код разбивается на логические блоки, каждый из которых выполняет какую-то определенную цель
3. Проще производить отладку кода

Как задавать функции

```
# Функция, которая выводит на экран 'Hello Ivan'
def say_hello():
    print('Hello Ivan')

say_hello()
```

- `def` – ключевое слово, даёт питону понять, что дальше будет задана функция
- `say_hello` – название функции, которое мы выбрали. Название может быть любым.
- Далее есть пара круглых скобок `()` . Внутри них могут быть разделенные запятыми параметры (что такое параметры будет рассмотрено далее).
- Затем идет двоеточие `:`, которое завершает строку определения функции.
- Дальше идет новая строка, начинающаяся с отступа (4 пробела или одно нажатие клавиши tab). В этой новой строке после отступа начинается тело функции: код действий, которые должна осуществлять функция при вызове.
- Наконец, в теле функции может быть опциональный оператор `return`, возвращающий значение вызывающей стороне при выходе из функции.

Жизнь функции состоит из двух этапов: объявление и вызов.

Объявление – это описание имени функции, всех входных значений, всех совершаемых функцией действий и возвращаемого результата.

Когда функция определена, код в ней не запускается сам по себе. Для его выполнения необходимо сделать вызов функции. Вызывать функцию можно столько раз, сколько вам нужно.

Для вызова функции используется следующий синтаксис:

```
function_name(arguments)
```

Сначала пишется имя функции. За ним пишутся круглые скобки. Если функция имеет обязательные аргументы, они должны быть перечислены в скобках (что такое аргументы будет рассмотрено далее).

Ключевое слово return

По умолчанию функция ничего не вернёт обратно при завершении своей работы. Чтобы она возвращала значение после вызова, добавьте ключевое слово `return` в теле функции перед переменной, которую хотите вернуть:

```
# Функция, которая определяет четное число или нет
def is_even(number):
    if number % 2 == 0:
        return True
    else:
```

```
        return False

print(is_even(2))
print(is_even(9))
```

> Аргументы функции

На предыдущем уроке был рассмотрен простой пример функции: ее задача была в том, чтобы вывести что-то на консоль. Но что, если в функцию нужно передать какие-то дополнительные данные?

В таком случае нужно ввести два дополнительных термина: параметры и аргументы.

Параметры — локальные переменные, которым присваиваются значения в момент ее вызова.

```
def say_hello(name):
    print(f'Hello {name}')
```

В этом примере параметром является `name`

В функцию можно передавать несколько аргументов, разделив их запятыми.

```
def name_and_age(name, age):
    print(f"I am {name} and I am {age} years old!")
```

В этом примере параметрами являются `name` и `age`.

При вызове функции в нее передаются аргументы.

Аргументы — это информация, переданная в функцию. Они представляют собой настоящие значения, соответствующие параметрам, которые были указаны при объявлении функции.

Таким образом, вызов функции `say_hello` будет выглядеть следующим образом:

```
say_hello("Ivan")
```

Аргументы по умолчанию

Аргументы функции могут иметь значения по умолчанию.

Чтобы аргумент функции имел значение по умолчанию, нужно назначить это значение параметру при определении функции.

Делается это в формате `ключ=значение`.

```
def say_hello(name="Ivan"):
    print(f"Hello {name})

# Выведет на консоль "Hello Ivan"
say_hello()
```

Вызывать такую функцию можно, не передавая ей никаких аргументов. В таком случае она будет использовать значение по умолчанию.

Больше информации

> Call Stack

Одна функция в ходе выполнения может вызывать другую функцию. В этом случае выполнение «внешней» функции приостановится, Python запомнит ее состояние и уйдет выполнять «внутреннюю» функцию. Когда «внутренняя» функция закончит выполняться, Python вернется к «внешней» функции и продолжит ее выполнять с того места, где остановился.

```
def inner_func(m):
    print("считаем a")
    a = m // 2
    a = a * a
    print("возвращаем a")
    return a

def outer_func(num):
    num += 2
    print(num)
    print("входим во внутреннюю функцию")
    k = inner_func(num)
    print("печатаем k")
    print(k, num)

outer_func(20)
```

Результатом выполнения функции будет:

```
22
входим во внутреннюю функцию
считаем a
возвращаем a
печатаем k
121 22
```

Больше информации

> Обработка ошибок в python

Очень важно уметь обрабатывать ошибки, когда речь заходит о построении надежных программ — каждый случай должен быть рассмотрен, и программа не должна завершаться с необработанной ошибкой.

Exceptions в Python не являются «ошибками» в традиционном понимании. Скорее это исключительные ситуации — когда программа не знает, что делать в возникшей ситуации, она выбрасывает исключение и прекращает свое выполнение, чтобы не навредить системе дальнейшими командами.

Базовый синтаксис

Для обработки ошибок в Python есть конструкция try/except. Общий шаблон выглядит следующим образом:

```
try:
    # В этом блоке могут быть ошибки
except <error type>:
    # Действия для обработки исключений;
    # Выполняется, если блок try выбрасывает ошибку
```

Блок try:

Это блок кода, который вы хотите выполнить. Этот блок может не работать должным образом, т.к во время выполнения из-за какого-нибудь исключения могут возникнуть ошибки.

Блок except:

Этот блок запускается, когда блок `try` не срабатывает из-за исключения. Если собираетесь перехватить ошибку как исключение, в блоке `except` нужно обязательно указать тип этой ошибки.

`Except` можно использовать и без указания типа ошибки, но так лучше не делать. В таком случае не учитывается, что возникшие ошибки могут быть разных типов. Вы будете знать, что что-то пошло не так, но что именно произошло — будет неизвестно.

Пример обработки ошибок:

```
try:
    1 / 0
except ZeroDivisionError:
    print('Попытались делить на ноль, я поймал ошибку и не дал пройти дальше')

print('Пишем строку после опасного кода, т.к программа не упала')
```

Список встроенных в Python ошибок можно найти в документации.

Обработка нескольких ошибок

В примере выше `except` ловил только одну ошибку. Но что делать, если необходимо обработать несколько ошибок? Объявлять несколько блоков `except`

```
a = []

try:
    # Так как элементов в списке нет, вылетит ошибка
    # Ошибка называется IndexError
    print(a[2])
    print(a[0] / a[1])
except IndexError:
    print('Такого элемента нет')
except ZeroDivisionError:
    print('Где-то делят на 0')
```

Больше информации

> Ссылочная модель данных

Мы уже выяснили, что функции могут принимать значения из других переменных себе на вход и проделывать какие-то операции над ними. Предположим, что мы редактируем переменную внутри функции. Что в таком случае будет с этими изменениями «снаружи» функции?

Пример с изменением переменной внутри функции:

Практический пример: в банке, где мы работаем, клиент может установить лимит трат на месяц. Пусть у нас есть список уже понесённых трат за месяц. Выставим проверку на каждую покупку клиента: в момент совершения покупки мысленно добавим ее сумму к списку трат, подсчитаем сумму и сравним с лимитом. Если получилось меньше — даем добро, иначе отклоняем. Подчеркнем: пока что покупка не совершена, это только прикидка.

```
def can_purchase(amount, history, limit):
    # Добавим в history покупку
    history.append(amount)
    # Затем просуммируем все элементы в history, сравним с limit и вернем True или False
    return sum(history) <= limit

limit = 100
history = [50, 40]

# Приходит параллельно два запроса на покупки (ни одну еще не совершил по факту)
# Должно разрешить покупку, т.к 90 + 4 <= 100
print(can_purchase(4, history, limit))
# Тоже должно разрешить покупку т.к 90 + 7 <= 100
print(can_purchase(7, history, limit))
```

Результатом работы программы будет:

```
True
False
```

Второй вызов вернул False, это что-то не то. Если посмотреть внимательно, то можно увидеть, что 90 + 4 + 7 уже больше 100 — ошибка может быть в этом.

Давайте выведем список:

```
def can_purchase(amount, history, limit, do_print=False):
    history.append(amount)
    if do_print:
        print(history)
    return sum(history) <= limit


limit = 100
client_history = [50, 40]

print(can_purchase(4, client_history, limit, do_print=True))  # Аргументы можно передавать по имени: явно говорим, что do_print будет равно
print(can_purchase(7, client_history, limit, do_print=True))
```

```
[50, 40, 4]
True
[50, 40, 4, 7]
False
```

Во втором вызове функции список содержал покупку из первого.

А если написать по-другому:

```
def can_purchase(amount, history, limit, do_print=False):
    local_copy = history.copy()  # работаем с копией history
    local_copy.append(amount)
    if do_print:
        print(local_copy)
    return sum(local_copy) <= limit


limit = 100
client_history = [50, 40]

print(can_purchase(4, client_history, limit, do_print=True))
print(can_purchase(7, client_history, limit, do_print=True))
```

```
[50, 40, 4]
True
[50, 40, 7]
True
```

Как же помогло использование .copy()? Это станет понятнее после следующего степа.

> Модель памяти в python

Чтобы ответить на вопрос из предыдущего степа, нужно сначала обратиться к модели памяти в Python.

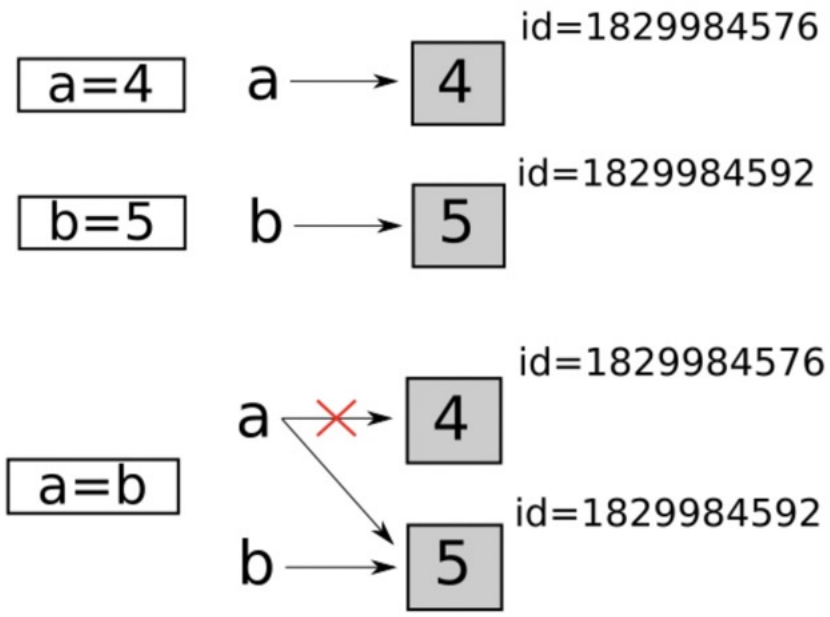
Начать нужно с того, что память компьютера линейна. Это значит, что данные в ней лежат длинным сплошным списком из нулей и единиц. Никаких двумерных матриц. Но мы уже знаем, что переменная позволяет записать некоторый объект в определенное имя, не задумываясь об устройстве памяти. Так мы записали в `history` список `[50, 40]`. Мы можем в него

добавлять элементы, удалять их — и не думать о линейности памяти и ее внутреннем устройстве. Как же этого можно добиться?

При инициализации переменной происходит примерно следующее:

1. В памяти компьютера создается объект (например, наш список). Можно представить, что в этот момент создается ячейка и объект кладется в эту ячейку.
2. Данный объект имеет некоторый идентификатор, значение и тип.
3. Где-то еще в памяти компьютера резервируется место под имя переменной, и в него кладется два значения: имя переменной и адрес в памяти, где должно лежать ее фактическое значение. В адрес памяти кладется фактический адрес созданного объекта.
4. Посредством оператора `=` создается ссылка между переменной и объектом.

Приведем пример, когда мы создаем две переменных `a` и `b`, а затем присваиваем переменную `b` в переменную `a`



Теперь разберем пример из прошлого степа. Когда мы на входе функции принимали аргумент `history`, мы по факту принимали указатель на список, который уже заранее был создан. Вызывая `.append()`, мы изменяли список «на месте» — добавляли элемент в тот же объект. После такого объект, на который ссылается `history`, изменяется навсегда — он был `[50, 40]`, а становится `[50, 40, 4]` И следующий вызов функции уже будет получать на вход ссылку, указывающую на список `[50, 40, 4]`.

Когда же мы делали `.copy()`, то фактически создавали новый объект в другом месте памяти, куда ушли все значения из `history`, и делали все изменения в новом объекте. Когда функция `can_purchase` завершилась, копия `history` уничтожилась — в конце выполнения функции все созданные в ней переменные уничтожаются.

Больше информации

> Изменяемые и неизменяемые типы данных

В Python существуют изменяемые и неизменяемые типы данных.

Изменяемые типы — под изменяемыми понимают типы, объекты которых могут быть изменены на месте.

Неизменяемые типы не дают себя менять, какими их создали в памяти, такими они и останутся до конца.

Как уже было сказано ранее, при создании переменной сначала создается объект, который имеет уникальный идентификатор, тип и значение, после этого переменная может ссылаться на созданный объект.

В случае с неизменяемыми типами созданный объект больше не изменяется. Например, если мы объявим переменную `k=15`, то будет создан объект со значением `15` типа `int`, и данный объект мы не сможем изменить. До конца он будет равным `15`.

Неизменяемые типы данных:

Целые числа (int)

Давайте определим переменную `x`, имеющую значение `10`. Встроенный метод `id()` используется для определения местоположения `x` в памяти, а `type()` используется для определения типа переменной. Когда мы пытаемся изменить значение `x`, оно успешно изменяется.

Стоит заметить, что адрес памяти тоже изменяется. Так происходит потому, что фактически мы не изменили значение `x`, а создали другой объект с тем же именем `x` и присвоили ему другое значение.

Строки (str)

То же самое верно и для строкового типа данных. Мы не можем изменить существующую переменную, вместо этого мы должны создать новую с тем же именем.

Кортежи (tuple)

Определим кортеж с 4 значениями. Воспользуемся функцией `id()` для вывода его адреса. Если мы захотим изменить значение первого элемента, то получим ошибку `TypeError`. Это означает, что кортеж не поддерживает присвоение или обновление элементов.

```
tuple1 = (1, 2, 3, 4)
print(tuple1, id(tuple1))
# Получим (1, 2, 3, 4) 3240603720336
```

Изменяемые типы данных

Списки (list)

Определим список с именем `x` и добавим в него некоторые значения. После этого обновим список: присвоим новое значение элементу с индексом 1. Можем заметить, что операция успешно выполнилась.

```
x = ['Яблоко', 'Груша', 'Слива']
x[1] = 'Ананас'
x   # выведет ['Яблоко', 'Ананас', 'Слива']
```

Вышеописанные действия являются простым и базовым примером модификации.

Словари (dict)

Словари — часто используемый тип данных в Python. Давайте посмотрим на их изменчивость.

Определим словарь под именем `dict` с тремя ключами и их значениями. Когда мы распечатаем его, отобразится все его содержимое. Можно распечатать каждое значение словаря отдельно, а также использовать ключи вместо индексов.

```
dict = {'Name':'Алиса', 'Age':27, 'Job':'Senior Python Developer'}
dict
# Получим {'Name': 'Алиса', 'Age': 27, 'Job': 'Senior Python Developer'}
dict['Name'], dict['Age'], dict['Job']
# ('Алиса', 27, 'Senior Python Developer')
```

Давайте изменим какое-нибудь значение в нашем словаре. Например, обновим значение для ключа `Name`. Выведем обновленный словарь. Значение изменилось. При этом сами ключи словаря неизменяемы.

```
dict['Name'] = 'Роберт'
dict   # {'Name': 'Роберт', 'Age': 27, 'Job': 'Senior Python Developer'}
```

Больше информации

> Хэш функции

Есть еще одна причина, почему неизменяемость важна. Для всех встроенных в Python неизменяемых объектов можно подсчитать хэш. Это свойство называется `_hashable_`, т.е. верно утверждение "tuple is hashable".

Хэш — это некая функция, которая берет на вход объект и считает одно число, причем для разных объектов это число разное. У хэш-функции есть два главных свойства:

1. Она быстро считается.
2. При малейшем изменении объекта хэш-функция меняется лавинообразно.

Хэш-функции позволяют организовать быстрый поиск и быстрое обращение по элементу, поэтому их использует «под капотом» словарь и множество. Собственно, из-за этого ключом в словаре не может выступать изменяемый объект (например, `list`) — для него нельзя подсчитать хэш. В прошлом уроке это просто проговорили, теперь же мы знаем причину.

Функция `hash()` возвращает хеш-значение объекта, если оно есть. Хэш-значения являются целыми числами.

```
hash('1')
-3723884734378080930 # Пример того, какое хэш-значение может иметь объект
```

Больше информации

> Срезы

Срезы позволяют обрезать список, взяв лишь те элементы, которые нужны. Они работают по следующей схеме: `list[НАЧАЛО:КОНЕЦ:ШАГ]`.

- Начало — с какого элемента стоит начать (по умолчанию равно 0);
- Конец — по какой элемент мы берем элементы (по умолчанию равно длине списка);
- Шаг — с каким шагом берем элементы, к примеру каждый 2 или 3 (по умолчанию каждый 1).

Рассмотрим примеры:

```
a = [1, 5, 8, 3, 4]
a[2:4]  # забрать элементы с третьего по пятый НЕ включительно (третий, четвертый)

# Результат
[8, 3]

# если опустить первый аргумент, то будет от начала списка
a[:4]  # по пятый НЕ включительно

# Результат
[1, 5, 8, 3]

# если опустить второй аргумент, то будет до конца списка
a[2:]

# Результат
[8, 3, 4]
```

```
# нумеровать можно отрицательными числами
# ниже описывается пример, как это будет считаться
# [1, 5, 8, 3, 4]
#  0  1  2  3  4
# -5 -4 -3 -2 -1
#a[1:len(a)-1]
a[1:-1]  # правый конец не включается, поэтому отдаст список со второго по предпоследний элемент

# Результат
[5, 8, 3]

# Может ничего не попасть
a[-1:-2]

# Результат
[]

a[-2:-1]

# Результат
[3]

# Есть еще третий аргумент - это шаг. Его можно пропустить
a[1:4:2]

# Результат
[5, 3]

a[::2]  # выдаст первый, третий и т.д.

# Результат
[1, 8, 4]

a[::-1]  # каждый "минус первый" - это каждый первый, только в обратном порядке, т.е. просто развернет лист

# Результат
[4, 3, 8, 5, 1]

# развернет и через один
a[::-2]

# Результат
[4, 8, 1]

a[3:1:-1]  # учтите, что в квадратных скобках индексы задаются от неразвернутого листа
# тут левый конец больше правого - при обратном порядке это нормально

# Результат
[3, 8]

# но при прямом это не сработает (от четвертого элемента до второго при движении вправо нет ничего)
a[3:1]

# Результат
[]
```

Больше информации

> Продвинутая работа со строками

В данном степе рассмотрим приемы работы со строками, которые чаще всего используются на практике.

Проверка символа

```
'l' in 'hello'
```

Результат:

```
True
```

Проверка подстроки

```
print('ll' in 'hello')
print('wl' in 'hello')
```

Результат:

```
True
False
```

Регистр

```
print('HeLlO'.lower())
print('hello'.upper())
```

Результат:

```
hello
HELLO
```

```
print('hello'.islower())
print('HELLO'.isupper())
```

Результат:

```
True
True
```

Замена части строки

```
'hello llevo'.replace('ll', 'mm')
```

Результат:

```
'hemmo mmevo'
```

Разбиение строки на пробелы

```
'Сегодня чудесный день'.split()
```

Результат:

```
['Сегодня', 'чудесный', 'день']
```

Убрать пробелы вокруг строки

```
'   После опознания текста много    пробелов     '.strip()
```

Результат:

```
'После опознания текста много    пробелов'
```

Больше информации