**Работа с функциями**

План лекции:

1. Функции.
2. Локальные переменные и глобальные переменные.
3. Функции с возвратом значения.
4. Применение функций.
5. **Функции**

Функция в программировании представляет собой обособленный участок кода, который можно вызывать, обратившись к нему по имени, которым он был назван. При вызове происходит выполнение команд тела функции.

Функции можно сравнить с небольшими программками, которые сами по себе, то есть автономно, не исполняются, а встраиваются в обычную программу. Нередко их так и называют – подпрограммы. Других ключевых отличий функций от программ нет. Функции также при необходимости могут получать и возвращать данные. Только обычно они их получают не с ввода (клавиатуры, файла и др.), а из вызывающей программы. Сюда же они возвращают результат своей работы.

Существует множество встроенных в язык программирования функций. С некоторыми из них в Python мы уже сталкивались. Это – print(), input(), int(), float(), str(), type(). Код их тела нам не виден, он где-то "спрятан внутри языка". Нам же предоставляется только интерфейс – имя функции.

С другой стороны, программист всегда может определять свои функции. Их называют пользовательскими. В данном случае под "пользователем" понимают программиста, а не того, кто использует программу. Разберемся, зачем нам эти функции, и как их создавать.

Предположим, надо три раза подряд запрашивать на ввод пару чисел и складывать их. С этой целью можно использовать цикл:

i = 0

while i < 3:

a = int(input())

b = int(input())

print(a+b)

i += 1

Однако, что если перед каждым запросом чисел, надо выводить надпись, зачем они нужны, и каждый раз эта надпись разная. Мы не можем прервать цикл, а затем вернуться к тому же циклу обратно. Придется отказаться от него, и тогда получится длинный код, содержащий в разных местах одинаковые участки:

print("Сколько бананов и ананасов для обезьян?")

a = int(input())

b = int(input())

print("Всего", a+b, "шт.")

print("Сколько жуков и червей для ежей?")

a = int(input())

b = int(input())

print("Всего", a+b, "шт.")

print("Сколько рыб и моллюсков для выдр?")

a = int(input())

b = int(input())

print("Всего", a+b, "шт.")

Внедрение функций позволяет решить проблему дублирования кода в разных местах программы. Благодаря им можно исполнять один и тот же участок кода не сразу, а только тогда, когда он понадобится.

**Определение функции. Оператор def.**

Итак, функция – отдельная, функционально независимая часть программы, выполняющая определенную задачу.

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 1 создание функции

В языке программирования Python функции определяются с помощью оператора def. Рассмотрим код:

def count\_food():

a = int(input())

b = int(input())

print("Всего", a+b, "шт.")

Это пример определения функции. Как и другие сложные инструкции вроде условного оператора и циклов, функция состоит из заголовка и тела. Заголовок оканчивается двоеточием и переходом на новую строку. Тело имеет отступ.

Ключевое слово def сообщает интерпретатору, что перед ним определение функции. За def следует имя функции. Оно может быть любым, также как и всякий идентификатор, например, переменная. В программировании желательно давать всему осмысленные имена. Так, в данном случае функция названа "посчитать\_еду" в переводе на русский.

После имени функции ставятся скобки. В приведенном примере они пустые. Это значит, что функция не принимает никакие данные из вызывающей ее программы. Однако она могла бы их принимать, и тогда в скобках были бы указаны так называемые параметры.

После двоеточия следует тело, содержащее инструкции, которые выполняются при вызове функции. Следует различать определение функции и ее вызов. В программном коде они не рядом и не вместе. Можно определить функцию, но ни разу ее не вызвать. Нельзя вызвать функцию, которая не была определена. Определив функцию, но ни разу не вызвав ее, вы никогда не выполните ее тела.

**Именование функций**

Имена функциям назначаются точно так же, как переменным. Имя функции должно быть достаточно описательным, чтобы любой, читающий ваш код, мог догадаться, что именно функция делает.

Python и тут требует соблюдения тех же правил, что при именовании переменных:

1. В имени функции используются только латинские буквы a-z, A-Z, цифры и символ нижнего подчеркивания (\_).

2. Имя функции не может начинаться с цифры.

3. Имя функции по возможности должно отражать ее назначение.

4. Символы верхнего и нижнего регистра различаются.

Поскольку функции выполняют действия, большинство программистов предпочитает в именах функций использовать глаголы. Например:

• функцию, которая рисует прямоугольник можно назвать draw\_box();

• функцию, которая печатает чек, можно назвать print\_check();

• функцию, которая вычисляет заработную плату до удержаний, можно назвать calculate\_gross\_рау().

Каждое из этих имен дает описание того, что функция делает.

**Вызов функции**

Рассмотрим полную версию программы с функцией:

def count\_food():

a = int(input())

b = int(input())

print("Всего", a+b, "шт.")

print("Сколько бананов и ананасов для обезьян?")

count\_food()

print("Сколько жуков и червей для ежей?")

count\_food()

print("Сколько рыб и моллюсков для выдр?")

count\_food()

После вывода на экран каждого информационного сообщения осуществляется вызов функции, который выглядит просто как упоминание ее имени со скобками. Поскольку в функцию мы ничего не передаем скобки опять же пустые. В приведенном коде функция вызывается три раза.

Когда функция вызывается, поток выполнения программы переходит к ее определению и начинает исполнять ее тело. После того, как тело функции исполнено, поток выполнения возвращается в основной код в то место, где функция вызывалась. Далее исполняется следующее за вызовом выражение.

В языке Python определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строка за строкой и о том, что находится ниже по, ему еще неизвестно. Поэтому, если вызов функции предшествует ее определению, то возникает ошибка (выбрасывается исключение NameError):

print("Сколько бананов и ананасов для обезьян?")

count\_food()

print("Сколько жуков и червей для ежей?")

count\_food()

print("Сколько рыб и моллюсков для выдр?")

count\_food()

def count\_food():

a = int(input())

b = int(input())

print("Всего", a+b, "шт.")

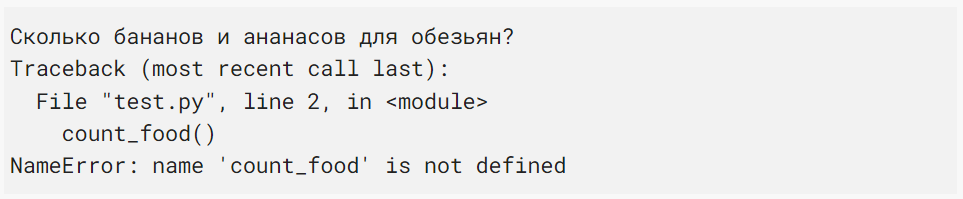


Рисунок 2 Результат неправильного объявления функции

**Функции с параметрами**

Функции с параметрами объявляются так же как функции без параметров, только с указанием в скобках:

def название\_функции(параметры):

блок кода

Изображение выглядит как текст

Автоматически созданное описание

Рисунок 3 создание функции с параметрами

Давайте напишем функцию draw\_box() так, чтобы она принимала параметры, задающие высоту и ширину прямоугольника:

def draw\_box(height, width): # функция принимает два параметра

for i in range(height):

print('\*' \* width)

Теперь наша функция draw\_box() принимает два целочисленных параметра height – высота прямоугольника и width – ширина прямоугольника, и для ее вызова нам нужно обязательно их указать.

Чтобы вывести звездный прямоугольник размерами 5 на 7 мы пишем код:

draw\_box(5, 7)

Результатом такого вызова функции draw\_box(5, 7) будет:

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

\*\*\*\*\*\*\*

На место параметров мы можем подставлять не только целочисленные константы, но и значения переменных. Следующий программный код:

n = 3

m = 9

draw\_box(n, m)

Для большей ясности давайте разберем еще один пример. Напишем функцию print\_hello(n), которая принимает одно натуральное число n и печатает слово Hello ровно n раз.

def print\_hello(n):

print('Hello' \* n)

Следующий программный код:

print\_hello(3)

print\_hello(5)

count = 2

print\_hello(count)

выведет:

HelloHelloHello

HelloHelloHelloHelloHello

HelloHello

Функцию print\_hello() можно сделать более гибкой, если передавать в нее еще один параметр – текст для вывода:

def print\_text(txt, n):

print(txt \* n)

Следующий программный код:

print\_text('Hello', 5)

print\_text('A', 10)

выведет:

HelloHelloHelloHelloHello

AAAAAAAAAA

**Что такое параметры и аргументы**

Аргумент – это любая порция данных, которая передается в функцию, когда функция вызывается.

Параметр – это переменная, которая получает аргумент, переданный в функцию.

Для функции draw\_box(height, width):

def draw\_box(height, width):

for i in range(height):

print('\*' \* width)

параметрами являются переменные height и width.

В момент вызова функции draw\_box(height, width):

height = 10

draw\_box(height, 9)

аргументами являются height и 9.

**Внесение изменений в параметры**

Когда аргумент передается в функцию, параметрическая переменная функции будет ссылаться на значение этого аргумента. Однако любые изменения, которые вносятся в параметрическую переменную, не будут влиять на аргумент.

Следующий программный код:

def draw\_box(height, width):

height = 2

width = 10

for i in range(height):

print('\*' \* width)

n = 5

m = 7

draw\_box(n, m)

print(n, m)

выведет:

\*\*\*\*\*\*\*\*\*\*

\*\*\*\*\*\*\*\*\*\*

5 7

В теле функции вносятся изменения в значения параметрических переменных height и width, однако это никак не повлияло на значение переменных n и m из основной программы, которые передавались в качестве аргументов в функцию draw\_box().

Так же:

1. Функции в Python могут принимать сколько угодно параметров.
2. Иногда вместо параметров и аргументов говорят о **формальных параметрах** и **фактических параметрах**. Формальные параметры – переменные, которые мы пишем при описании функции. Фактические параметры – то, что реально подставляется при вызове функции.

**2. Локальные переменные и глобальные переменные**

**Локальные переменные**

Локальными называются переменные, объявленные внутри функции и доступные только ей самой. Программный код за пределами функции к ним доступа не имеет.

Рассмотрим функцию print\_texas(), которая выводит информацию о количестве птиц, обитающих в Техасе.

def print\_texas():

birds = 5000

print('В Техасе обитает', birds, 'птиц.')

В теле функции мы создаем переменную birds, которой присваивается значение, равное 5000. Такая переменная является локальной для функции print\_texas(). Всякий раз, когда переменной внутри функции присваивается значение, в результате создается локальная переменная. Она принадлежит функции, в которой создается, и к ней получает доступ только программный код этой функции.

Термин «локальная» указывает на то, что переменная может использоваться лишь в этом месте — внутри функции, в которой создается.

Если программный код одной функции попытается обратиться к локальной переменной, принадлежащей другой функции, произойдет ошибка.

Рассмотрим следующий программный код:

def print\_texas():

birds = 5000

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

print('В Калифорнии обитает', birds, 'птиц.')

Функция print\_california() обращается к локальной переменной birds функции print\_texas(). Вызов функции print\_california(), приводит к ошибке:

NameError: name 'birds' is not defined

Локальные переменные скрыты от других функций, поэтому другие функции могут иметь собственные локальные переменные с тем же именем. Например,

def print\_texas():

birds = 5000

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

birds = 9000

print('В Калифорнии обитает', birds, 'птиц.')

В каждой из этих двух функций есть локальная переменная с именем birds. Но они никогда не видны одновременно, так как находятся в разных функциях.

Когда выполняется функция print\_texas(), видима переменная birds, значение которой равно 5000. Когда выполняется функция print\_california(), видима переменная birds, значение которой равно 9000.

**Глобальные переменные**

Глобальными называются переменные, объявленные в основной программе и доступные как программе, так и всем ее функциям.

Рассмотрим следующий программный код:

birds = 5000 # глобальная переменная

def print\_texas():

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

print('В Калифорнии обитает', birds, 'птиц.')

В самом начале программы создаем глобальную переменную birds, значение которой равно 5000. Далее описываем две функции, обращающиеся к глобальной переменной. Результатом выполнения следующего кода:

print\_texas()

print\_california()

будет:

В Техасе обитает 5000 птиц.

В Калифорнии обитает 5000 птиц.

Функция может использовать любые глобальные переменные кроме имеющих те же имена, что и ее локальные переменные. Если в функции объявлена локальная переменная с тем же именем, что у одной из глобальных, то данная глобальная переменная становится недоступной в этой функции, и при указании идентификатора переменной произойдет обращение к локальной переменной функции, а не одноименной глобальной.

Рассмотрим следующий программный код:

birds = 5000 # глобальная переменная

def print\_texas():

birds = 1000 # локальная переменная

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

birds = 7000 # локальная переменная

print('В Калифорнии обитает', birds, 'птиц.')

В самом начале программы мы создаем глобальную переменную birds, значение которой равно 5000. Далее мы описываем две функции, в которых создаются локальные переменные с таким же именем birds. Таким образом при обращении к переменной birds внутри функций будет происходить обращение именно к локальной переменной.

Результатом выполнения следующего кода:

print\_texas()

print\_california()

будет:

В Техасе обитает 1000 птиц.

В Калифорнии обитает 7000 птиц.

**Ключевое слово global**

Если нужно, чтобы инструкция внутри функции присваивала значение глобальной переменной, то требуется дополнительный шаг. В этом случае, глобальная переменная должна быть объявлена внутри функции.

Рассмотрим следующий программный код:

def print\_texas():

global birds

birds = 5000

print('В Техасе обитает', birds, 'птиц.')

def print\_california():

print('В Калифорнии обитает', birds, 'птиц.')

print\_texas()

print\_california()

Результатом выполнения следующего кода:

print\_texas()

print\_california()

будет:

В Техасе обитает 5000 птиц.

В Калифорнии обитает 5000 птиц.

**3. Функции с возвратом значения**

Функция с возвратом значения похожа на функцию без возврата значения тем, что:

• это набор инструкций, выполняющий определенную задачу;

• когда нужно выполнить функцию, ее вызывают.

Однако, когда функция с возвратом значения завершается, она возвращает значение в ту часть программы, которая ее вызвала. Возвращаемое из функции значение используется как любое другое: оно может быть присвоено переменной, выведено на экран, использовано в математическом выражении (если это число) и т. д.

Мы уже сталкивались со многими функциями с возвратом значений:

• функция int() – преобразует строку к целому числу и возвращает его;

• функция float() – преобразует строку к вещественному числу и возвращает его;

• функция range() – возвращает последовательность целых чисел 0, 1, 2, ...;

• функция abs() – возвращает абсолютное значение числа (модуль числа);

• функция len() – возвращает длину строки или списка.

Функцию с возвратом значения пишут точно так же, как и без, но она должна иметь инструкцию return.

Вот общий формат определения функции с возвратом значения в Python:

def название\_функции():

блок кода

return выражение

В функции должна быть инструкция return, принимающая форму:

return выражение

Значение выражения, которое следует за ключевым словом return, будет отправлено в ту часть программы, которая вызвала функцию. Это может быть переменная либо выражение, к примеру, математическое.

Давай те решим задачу перевода градусов по шкале Фаренгейта в градусы по шкале Цельсия по формуле C = 5 / 9 \* (F− 32).

Напишем функцию, которая осуществляет перевод:

def convert\_to\_celsius(temp):

result = (5 / 9) \* (temp - 32)

return result

Задача этой функции — принять одно число temp в качестве аргумента – количество градусов по шкале Фаренгейта, и вернуть другое — количество градусов по шкале Цельсия.

Рассмотрим ее работу. Первая инструкция в блоке функции присваивает значение (5 / 9) \* (temp - 32) переменной result. Затем исполняется инструкция return, которая приводит к завершению исполнения функции и отправляет значение из переменной result, назад в ту часть программы, которая вызвала эту функцию.

# функция перевода градусов Фаренгейта в градусы Цельсия

def convert\_to\_celsius(temp):

result = (5 / 9) \* (temp - 32)

return result

# основная программа

temp = float(input('Bвeдитe количество градусов по Фаренгейту: '))

celsius = convert\_to\_celsius(temp)

print(celsius) # градусы Цельсия

Основная программа получает от пользователя одно число – значение в градусах Фаренгейта, и вызывает функцию, передавая значение переменной temp в качестве аргумента. Значение, которое возвращается из функции convert\_to\_celsius, присваивается переменной celsius.

Взглянем еще раз на функцию convert\_to\_celsius():

def convert\_to\_celsius(temp):

result = (5 / 9) \* (temp - 32)

return result

Обратите внимание, что внутри этой функции происходят две вещи: во-первых, переменной result присваивается значение выражения (5 / 9) \* (temp - 32), и во-вторых, значение переменной result возвращается из функции. Эта функция хорошо справляется с поставленной перед ней задачей, но ее можно упростить. Поскольку инструкция return возвращает значение выражения, переменную result устраняем и переписываем функцию так:

def convert\_to\_celsius(temp):

return (5 / 9) \* (temp - 32)

Эта версия функции не сохраняет значение (5 / 9) \* (temp - 32) в отдельной переменной, а сразу возвращает значение выражения с помощью инструкции return. Делает то же, что и предыдущая версия, но за один шаг.

**Использование нескольких return**

В одной функции может быть сколько угодно инструкций return. Рассмотрим функцию convert\_grade(), которая переводит стобалльную оценку в пятибалльную:

def convert\_grade(grade):

if grade >= 90:

return 5

elif grade >= 80:

return 4

elif grade >= 70:

return 3

elif grade >= 60:

return 2

else:

return 1

# основная программа

grade = int(input('Введите вашу отметку по 100-балльной системе: '))

print(convert\_grade(grade))

В функции convert\_grade() используется 5 инструкций return. Каждая из них возвращает соответствующее значение и завершает работу функции.

Функцию convert\_grade() можно переписать с помощью одной инструкции return:

def convert\_grade(grade):

result = -1

if grade >= 90:

result = 5

elif grade >= 80:

result = 4

elif grade >= 70:

result = 3

elif grade >= 60:

result = 2

else:

result = 1

return result

Функции с возвратом значения предоставляют те же преимущества, что функции без возврата значения:

* упрощают программный код;
* уменьшают дублирование кода;
* упрощают тестирование кода;
* увеличивают скорость разработки;
* способствуют работе в команде.

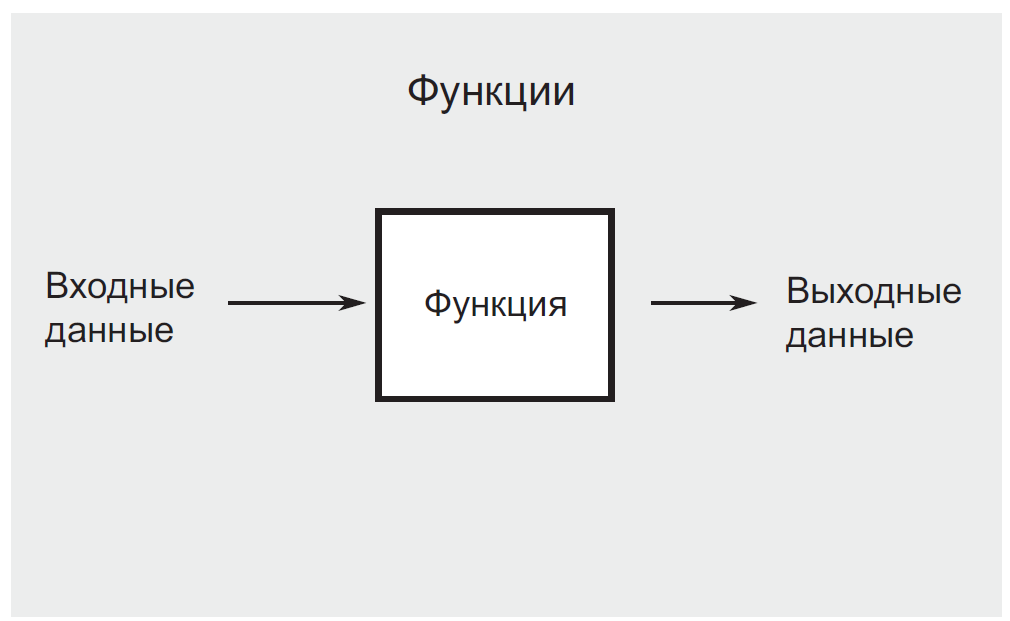


Рисунок 4 Графическая интерпретация работы функции с возвратом значения

**Возвращение булевых значений**

Python позволяет писать булевы функции, возвращающие либо истину (True), либо ложь (False). Булеву функцию можно применять для проверки условия, тогда значения True и False будут сигнализировать о его выполнении.

Булевы функции широко применяются для упрощения сложных условий, проверяемых в структурах принятия решения и структурах с повторением.

Например, напишем программу, которая просит пользователя ввести число, и определяет четное оно или нечетное.

Это можно сделать так:

number = int(input())

if number % 2 == 0:

print('Это число четное. ')

else:

print('Это число нечетное.')

Этот фрагмент кода будет легче понять, если написать булеву функцию is\_even(), которая принимает число в качестве аргумента и возвращает True, если оно четное, и False если нечетное.

def is\_even(number):

if number % 2 == 0:

return True

else:

return False

Теперь можно переписать инструкцию if-else основной программы так, чтобы она для определения четности переменной number вызывала функцию is\_even():

number = int(input())

if is\_even(number):

print('Это число четное. ')

else:

print('Это число нечетное.')

Так логику программы легче понять, а функцию можно вызывать в программе всякий раз, когда необходимо проверить четность числа.

**Использование булевых функций для валидации входных данных**

Булевы функции можно также использовать для упрощения сложного кода валидации входных данных. Например, в программе, предлагающей пользователю ввести номер модели изделия, где возможны только значения 100, 200 и 300, можем написать такой код:

model = int(input())

while model != 100 and model != 200 and model != 300:

рrint('Допустимыми номерами моделей являются 100, 200 и 300.')

model = int(input())

Цикл валидации использует длинное составное булево выражение, и повторяется до тех пор, пока model не будет равняться 100 и 200 или 300.

Вместе с тем, цикл валидации можно упростить, написав булеву функцию проверки переменной model, и вызывая ее в цикле. Напишем функцию is\_invalid(), которая принимает один параметр model и возвращает значение True, если модель недопустима и False в противном случае. Тогда цикл валидации можно переписать следующим образом:

while is\_invalid(model):

print('Допустимыми номерами моделей являются 100, 200 и 300.')

model = int(input())

После этого изменения цикл становится легче читать. Теперь вполне очевидно, что цикл повторяется до тех пор, пока номер модели недопустим. Приведенный ниже фрагмент кода показывает, как можно было бы написать функцию is\_invalid(). Она принимает номер модели в качестве аргумента, и если аргумент не равен 100, 200 и 300, то эта функция возвращает True, говоря, что он недопустимый. В противном случае функция возвращает False.

def is\_invalid(model):

if model != 100 and model != 200 and model != 300:

return True

else:

return False

**Функции с возвратом нескольких значений**

В Python функции не ограничены возвратом всего одного значения. После инструкции return можно определить много выражений, разделенных запятыми:

return выражение 1, выражение 2, выражение 3 ...

Следующий программный код определяет функцию get\_powers(num), которая принимает в качестве аргумента число num и возвращает его квадрат, куб и четвертую степень.

def get\_powers(num):

return num\*\*2, num\*\*3, num\*\*4

Результатом выполнения следующего кода:

a, b, c = get\_powers(2)

print(a)

print(b)

print(c)

будет:

4

8

16

**4. Применение функций**

Факториал числа:

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

result = factorial(5)

print(result) # Вывод: 120

Проверка палиндрома:

def is\_palindrome(word):

word = word.lower()

return word == word[::-1]

print(is\_palindrome("radar")) # Вывод: True

print(is\_palindrome("python")) # Вывод: False

Генерация случайного пароля:

import random

import string

def generate\_password(length):

characters = string.ascii\_letters + string.digits + string.punctuation

password = ''.join(random.choice(characters) for \_ in range(length))

return password

password = generate\_password(10)

print(password) # Вывод: pY5r&@S^b9

Камень, ножницы, бумага:

import random

def get\_user\_choice():

user\_choice = input("Выберите: камень (r), ножницы (s), бумага (p): ")

if user\_choice.lower() in ['r', 's', 'p']:

return user\_choice.lower()

else:

print("Некорректный выбор. Попробуйте снова.")

return get\_user\_choice()

def get\_computer\_choice():

choices = ['r', 's', 'p']

return random.choice(choices)

def determine\_winner(user\_choice, computer\_choice):

if user\_choice == computer\_choice:

return "Ничья!"

elif (

(user\_choice == 'r' and computer\_choice == 's') or

(user\_choice == 's' and computer\_choice == 'p') or

(user\_choice == 'p' and computer\_choice == 'r')

):

return "Вы победили!"

else:

return "Компьютер победил!"

def play\_game():

print("Игра Камень, ножницы, бумага началась!")

user\_choice = get\_user\_choice()

computer\_choice = get\_computer\_choice()

print(f"Вы выбрали: {user\_choice}")

print(f"Компьютер выбрал: {computer\_choice}")

print(determine\_winner(user\_choice, computer\_choice))

play\_game()

**Преимущества использования функций**

Разбивая программы на функции получаем:

• Более простой код. Разбитый на функции код программы проще и легче для понимания. Несколько небольших функций намного легче читать, чем одну длинную последовательность инструкций;

• Повторное использование кода. Функции позволяют избежать многократного повторения кода в программе. Если какая-то операция в программе выполняется в нескольких местах, то можно один раз написать для нее функцию и затем исполнять ее, когда понадобится.

• Более простое тестирование. Когда каждая задача в программе содержится в собственной функции, программисты могут индивидуально протестировать каждую функцию и определить, выполняет ли она свою задачу правильно.

• Более быстрая разработка. Предположим, что программист или команда программистов разрабатывает много программ. Они обнаруживают общие задачи у разных программ, например выяснение имени пользователя и пароля, вывод текущего времени. Каждый раз писать программный код для этих задач не имеет смысла. Для часто встречающихся задач пишут функции, и включают в состав любой нуждающейся в них программы.

• Упрощение командной работы. Когда программа разрабатывается как набор функций, разным программистам можно поручить написание отдельных функций.