Функции

1 Объявление функций

Сегодня мы поговорим про функции в Python. Представим, что вам в вашей программе часто нужно выполнять одинаковые действия с разными переменными. Например, допустим на вход передается два списка, и мы хотим в каждом из них найти максимум – нам придется в двух местах программы писать одинаковый код с условным оператором:

```
a = [1, 3, 4]
b = [9, 3, 6, 29]

max_a = a[0]            # поиск мксимума в первом массиве
for x in a:
  if x > max_a:
    max_a = x

max_b = b[0]            # поиск мксимума во втором массиве
for x in b:
  if x > max_b:
    max_b = x

print(max_a)
print(max_b)
```

В более сложных программах вычислять максимум в массиве приходится еще чаще. При этом чем больше мы пишем или копируем кода, тем у нас больше шансов допустить ошибку. Чтобы избежать повторного написания одной и той же логики, в языках программирования существуют **функции**.

Функции — это такие участки кода, которые изолированы от остальной программы и выполняются только тогда, когда вызываются. Вы уже встречались с функциями `ceil()`, `len()` и `print()`. Функции обладают общим свойством: они могут принимать параметры (ноль, один или несколько) и возвращать значение (а могут и не возвращать). Давайте посмотрим на пример функции, которая возвращает максимум из двух чисел. Функция `max` принимает два параметра-числа, максимум из которых требуется определить, а возвращает одно число:

```
def max(a, b):
  if a > b:
    return a
  return b
```

Код функции должен размещаться в начале программы – до того места, где мы захотим воспользоваться функцией. Первая строчка этого примера является описанием нашей функции. Объявление новой функции начинается с ключевого слова `def`. `max` — идентификатор, имя нашей функции. Правила для записи имени функции такие же, как и для переменных: имя может содержать строчные буквы английского алфавита, цифры и знаки подчёркивания. После идентификатора в круглых скобках идет список параметров, которые получает наша функция. Список состоит из перечисленных через запятую идентификаторов параметров. В нашем случае список состоит из двух переменных `a` и `b`. В конце строки ставится двоеточие.

Далее идет тело функции, оформленное с отступом, в виде блока. Внутри функции вычисляется значение максимума, которое возвращается из функции с помощью оператора `return`. После достижения оператора `return` функция заканчивает свое выполнение и программа возвращается в место вызова этой функции. Если функция не возвращает значения, то инструкция `return` используется без возвращаемого значения. В функциях, которым не нужно возвращать значения, инструкция `return` может и отсутствовать.

Давайте теперь посмотрим на использование этой функции. Мы можем передавать в качестве параметров как константные значения, так и переменные, и элементы контейнеров (например, списков):

```
def max(a, b):
  if a > b:
    return a
  return b

print(max(3, 7))

a = 10
b = 2
m = max(a, b)
print(m)

numbers = [3, 4, 5]
max_element = numbers[0]
for x in numbers:
  max_element = max(max_element, x)
print(max_element)
```

```
7
10
5
```

В результате первого вызова функции `max` (вместе с предоставленной нам Python

функцией `print`) будет напечатано `7`, во втором случае в переменную `m` будет записано наибольшее из значений `a` и `b`, а в третьем случае переменная `max_element` будет содержать наибольший элемент списка `numbers`, так как мы последовательно сравним каждый элемент списка с текущим максимальным значением.

Разберем чуть более сложный пример:

Задача

Дана матрица размера $n \times m$. Требуется найти в ней строку, в которой число с максимальной суммой цифр минимально (гарантируется, что в каждой строке число с максимальной суммой цифр единственное).

Входные данные	*Выходные данные*
33 606 31 106 10000000 14 1 150 91	2

В приведенном премере, в строке с индексом 2 число с максимальной суммой цифр – это 91 *(* $9+1>1+5+0>1$*), что меньше, чем* 606 *в строке* 0 *и* 106 *в строке* 1*.*

Чтобы решить эту задачу, для каждого числа в матрице необходимо посчитать сумму его цифр, в каждой строке найти число с максимальной суммой цифр, а затем выбрать такую строку, в которой найденное число минимально.

Итак, напишем сначала функцию для нахождения суммы цифр числа. Эта функция, которую мы назовем `calc_sum_digits`, будет принимать одно число `x` и возвращать тоже одно число, равное сумме цифр `x`. Создадим промежуточную переменную `s` для суммы цифр, а дальше разобьем число на цифры, как мы делали в одном из прошлых занятий, с помощью цикла `while`:

```
def calc_sum_digits(x):
  s = 0
  while (x > 0):
    s += x % 10
    x /= 10
  return s
```

К моменту, когда `x` станет равным нулю, переменная `s` как раз накопит сумму всех цифр числа – её мы и возвращаем в функции.

Теперь напишем функцию, которая находит в строке матрицы число с максимальной суммой цифр. Функция `find_maxdigitsum_element` будет принимать список (строку матрицы), и возвращать искомое число. Пройдемся по списку, и для каждого числа вызовем функцию `calc_sum_digits`, которая вернет сумму цифр числа. Если эта сумма окажется больше, чем текущий максимум, то обновим и значение максимальной суммы в переменной `max_sum`, и индекс найденного максимума, который хранится в переменной `max_index`. После обработки всего списка, вернем число, которое находится на позиции индекса числа с максимальной суммой цифр, то есть элемент списка на позиции `max_index`:

```
def find_maxdigitsum_element(row):
  max_sum = -1
  max_index = -1
  for i in range(len(row)):
    cur_sum = calc_sum_digits(row[i])
    if cur_sum > max_sum:
      max_sum = cur_sum
      max_index = i
  return row[max_index]
```

В основной части программы нам остается только вызвать функцию `find_maxdigitsum_element` для каждой строки матрицы и сравнить возвращаемое функцией значение с текущим минимумом:

```
matrix = [[33, 606, 31], [106, 10000000, 14], [1, 150, 91]]
min_value = find_maxdigitsum_element(matrix[0])
row = 0

for i in range(len(matrix)):
  max_sum = find_maxdigitsum_element(matrix[i])
  if max_sum < min_value:
    min_value = max_sum
    row = i
print(row)
```

```
2
```

Обратите внимание, что на момент вызова функции, она уже должна быть реализована в программе, то есть описание функции должно находиться выше по коду. Это должно выполняться как для функции `calc_sum_digits`, так и для функции `find_maxdigitsum_element`.

В данном примере, функции вызываются всего в одном месте, поэтому не сокращают длину программы, а наоборот, увеличивают. Однако разбиение длинного кода на небольшие кусочки, объединенные общей логикой, сильно упрощают написание и понимания кода. В такой программе проще искать ошибку – вы всегда можете протестировать каждую функцию в отдельности, а также вам проще понимать, что происходит в том или ином фрагменте кода. Чем сложнее будут становиться задачи, тем острее будет нужда в хорошем разбиении на функции.

2 Переменные и фунции

2.1 Локальные переменные

Переменные, объявленные внутри функции, видны только внутри этой функции. Рассмотрим пример:

```
def example():
    a = 10

example()
print(a)
```

В данном примере переменной `a` присваивается значение внутри функции `example`, эта функция вызывается, а затем программа пытается обратиться к значению этой переменной. В результате запуска этой программы мы получим ошибку *NameError: name 'a' is not defined*. Такие переменные, объявленные внутри функции, называются локальными. Эти переменные становятся недоступными после выхода из функции.

2.2 Глобальные переменные

Внутри функции можно использовать переменные, объявленные вне этой функции:

```
def example():
    print(a)

a = 10
example()
```

```
10
```

В этом примере переменной `a` присваивается значение `10`, и функция `example` печатает это значение несмотря на то, что до объявления функции эта переменная не инициализируется. В момент вызова функции переменной `a` уже присвоено значение, поэтому функция `example` может вывести его на экран. Такие переменные (объявленные вне функции, но доступные внутри) называются глобальными.

2.3 Изменение глобальных переменных внутри функции

Интересным получится результат, если попробовать изменить значение глобальной переменной внутри функции. Рассмотрим следующую программу:

```
def example():
  a = 1
  print(a)

a = 0
example()
print(a)
```

```
1
0
```

При запуске выводится `1 0`. Несмотря на то, что значение переменной `a` изменилось внутри функции, вне функции оно осталось прежним. Это сделано в целях "защиты"

глобальных переменных от случайного изменения внутри функции. Например, если функция будет вызвана из цикла по переменной `i`, а в этой функции тоже будет использована переменная `i` (стандартное название переменной для цикла `for`), то эти переменные должны быть различными – изменение внутри функции не должно менять значение снаружи.

Таким образом, если внутри функции модифицируется значение некоторой переменной, то переменная с таким именем становится локальной переменной, и ее модификация не приведет к изменению глобальной переменной с таким же именем. Чтобы изменить глобальную переменную, а не создавать локальную копию, необходимо объявить эту переменную внутри функции, как глобальную, при помощи ключевого слова `global`:

```
def example():
  global a
  a = 1
  print(a)

a = 0
example()
print(a)
```

```
1
1
```

Теперь наш код выводит `1 1` – функция меняет значение глобальой переменной.

Тем не менее, лучше не изменять значения глобальных переменных внутри функции. Когда функции необходимо изменить переменную, лучше всего реализовать это так: пусть функция вовращает новое значение с помощью `return`, после чего мы явно присвоим возвращаемое значение в нужную переменную уже снаружи функции:

```
def example():
    new_a = 1
    print(new_a)
    return new_a

a = 0
a = example()
print(a)
```

```
1
1
```

Если следовать этому правилу, то функции получаются независимыми от глобальных переменных и всего остального кода, и их можно легко копировать из одной программы в другую. Такие правила приняты у программистов, но в олимпиадном программировании для упрощения логики иногда будет удобно воспользоваться глобальными переменными. Однако в рамках этого курса нам это не понадобится.

2.4 Передача списков в аргументы функции

Все контейнеры, в том числе списки, передаются в аргументы функции по ссылке (то есть копирования не происходит). Если мы изменим значение элемента в списке, добавим в него элемент или удалим элемент, то все эти изменения произойдут со списком, переданным в функцию в качестве аргумента. Если вы хотите скопировать список, а не передавать его по ссылке, то при передаче его в функцию можете воспользоваться срезами:

```
def example(a):
  for i in range(len(a)):
    a[i] += 1

first = [1, 2]
second = [1, 2]

example(first)      # по умолчанию, список не копируется
example(second[:])  # передавая срез, создаем копию исходного списка

print(first)
print(second)
```

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

В данной программе, список `first` изменился после выполнения функции (список был передан по ссылке), а список `second` не изменился (список был скопирован).

3 Возвращаемые значения функций

Функция может возвращать несколько значений. Для этого можно вернуть кортеж. Рассмотрим пример функции, которая находит наибольшее четное и нечетное значения в списке. Мы хотим вернуть оба этих значения. Вернем кортеж из них, тогда результат вызова функции можно будет использовать во множественном присваивании:

```
def max_even_odd(a):
  even = -1
  odd = -1
  for x in a:
    if x % 2 == 0:
      even = max(even, x)
    else:
      odd = max(odd, x)
  return even, odd            # возвращаем кортеж из двух чисел

even, odd = max_even_odd(test)
```

4 Аргументы функции

4.1 Значение по умолчанию для аргументов функции

В Python есть возможность задать для аргументов функции значение по умолчанию. Если значение для такого аргумента при вызове функции не передаётся, то используется значение по умолчанию. Пусть у нас есть функция, которая увеличивает все элементы списка на некоторое `x`. Если `x` не будет передан, то по умолчанию возьмется значение `1`:

```
def increment(mylist, x=1):
  for i in range(len(mylist)):
    mylist[i] += x

test = [1, 10, 3, 4, 5, 5, 5, 7]
increment(test)     # аргумент не передан, берется значение по умолчанию x = 1
print(test)
increment(test, -1) # передан аргумент, x = -1
print(test)
```

```
[2, 11, 4, 5, 6, 6, 6, 8]
[0, 9, 2, 3, 4, 4, 4, 6]
```

4.2 Передача значений аргументов по имени

Часто в функции передаётся большое количество аргументов, и вспомнить порядок их перечисления в функции бывает сложно. В Python есть возможность передать значение аргумента по его имени. В таком случае аргумент становится уже не позиционным, а именованным. Именованному аргументу присваивается значение при вызове функции:

```
def increment(mylist, x=1):
  for i in range(len(mylist)):
    mylist[i] += x

test = [1, 10, 3, 4, 5, 5, 5, 7]
increment(test)
print(test)
increment(test, -1)
print(test)
increment(x=3, mylist=test)
print(test)
increment(test, x=-1)
print(test)
```

Обратите внимание на порядок перечисления аргументов в функции: сначала передаются позиционные аргументы, а уже потом именованные. Для именованных аргументов порядок перечисления при вызове функции и её объявлении может не совпадать.

Вы уже наверняка сталкивались с именованными аргументами в функции `print`. У неё есть два популярных именованных аргумента, каждый из которых имеет значение по умолчанию: `sep` и `end`. Функция `print` принимает несколько позиционных аргументов – объекты, которые требуется вывести, аргумент `sep`, равный по умолчанию пробелу, задает разделитель между переданными объектами, а аргумент `end`, равный по умолчанию переводу строки, задает строку, которая выводится в конце вывода:

```
s = '123'
print(6, 78, s, sep='XX', end=':)')
```

```
6XX78XX124:)
```