

Углубленный курс информатики

Лекция 5
Функции

5 марта 2022 г.

Вячеслав Алексеевич Чузлов,
к.т.н., доцент ОХИ ИШПР

Содержание

1. Описание функций
 - Оператор `def`
 - Оператор `return`
 - Параметры функций
 - Локальные переменные
 - Оператор `global`
 - Области видимости и вложенные функции
 - Замыкания
 - Оператор `nonlocal`
 - Строки документации (`docstring`)
2. Синтаксис передачи аргументов
 - Именованные аргументы и значения по умолчанию
 - Передача произвольного количества аргументов
3. Рекурсивные функции

Описание функций

Написание кода функций

- **Функции** – это многократно используемые фрагменты программы. Они позволяют дать имя определенному блоку команд с тем, чтобы в последствии запускать блок по указанному имени в любом месте программы и сколь угодно много раз. Это называется *вызовом функции*.
- Функции определяются при помощи зарезервированного слова `def`. После этого слова указывается имя функции, за которым следует пара скобок, в которых можно указать имена некоторых переменных, и заключительное двоеточие в конце строки. Далее следует блок команд (инструкций), составляющих тело функции.
- **Сигнатура функции** – часть общего объявления функции, позволяющая средствами трансляции идентифицировать функцию среди других. Составляющие сигнатуры:
 1. имя функции;
 2. аргументы функции;
 3. возвращаемые значения.

```
1 >>> def say_hi():
2     ...     print('Hi!')
3
4 >>> say_hi()
5 Hi!
6
```

Оператор `def`

- Оператор `def` создает объект функции и присваивает его имени.
- Общий формат оператора `def` выглядит следующим образом:

```
def имя(аргумент1, аргумент2, ... аргументN):  
    операторы
```

- В строке заголовка `def` указывается имя функции, которому присваивается объект функции, а также список из нуля и более аргументов (иногда называемых параметрами) в круглых скобках.
- Именам аргументов в заголовке присваиваются объекты, передаваемые в круглых скобках при вызове функции.

Оператор `def`

- Тело функции почти всегда содержит оператор `return`:
`def имя(аргумент1, аргумент2, ... аргументN):`
 операторы
 ...
 `return` значение
- Оператор `return` в Python может появляться где угодно в теле функции; по достижении он заканчивает выполнение функции и возвращает результат обратно вызывающему коду.
- Оператор `return` состоит из необязательного выражения с объектным значением, которое дает результат функции.
- Если значение опущено, тогда `return` возвращает `None`.
- Оператор `return` сам по себе также необязателен; если он отсутствует, то выход из функции происходит, когда интерпретатор достигает конца тела функции. Формально функция без оператора `return` автоматически возвращает объект `None`.
- Хорошим тоном является явное использование пустого оператора `return` для дополнительного пояснения того, что функция ничего не возвращает в качестве результата.

Оператор `return`

- Оператор `return` используется для возврата⁵ из функции, т.е. для прекращения её работы и выхода из неё. При этом можно также вернуть некоторое значение из функции.
- Оператор `return` в Python может появляться где угодно в теле функции; по достижении он **заканчивает выполнение функции** и возвращает результат обратно вызывающему коду.

```
1 >>> def maximum(x, y):
2 ...     if x > y:
3 ...         return x
4 ...     elif x == y:
5 ...         return "Equals."
6 ...     else:
7 ...         return y
8
9 >>> print(maximum(2, 3))
10 3
11
```

```
1 >>> def maximum(x, y):
2 ...     if x > y:
3 ...         return x
4 ...     if x == y:
5 ...         return "Equals."
6 ...     return y
7
8 >>> print(maximum(2, 3))
9 3
10
```

Параметры функций

- Функции могут принимать параметры, т.е. некоторые значения, передаваемые функции для того, чтобы она что-либо сделала с ними.
- Эти параметры похожи на переменные, за исключением того, что значение этих переменных указывается при вызове функции, и во время работы функции им уже присвоены их значения.
- Параметры указываются в скобках при объявлении функции и разделяются запятыми. Аналогично мы передаём значения, когда вызываем функцию.
- Обратите внимание на терминологию: имена, указанные в объявлении функции, называются **параметрами**, тогда как значения, которые Вы передаёте в функцию при её вызове, – **аргументами**.

Примеры определения и вызова функций

Определение

Ниже показано определение функции по имени `times`, которое возвращает произведение двух аргументов:

```
1 | def times(x, y):  
2 |     return x * y  
3 |
```

- Когда интерпретатор встречает и выполняет этот оператор `def`, он создает новый объект функции, вмещающий в себе код функции, и присваивает его имени `times`.
- Обычно такой оператор находится в файле модуля и выполняется при его импортировании.

Примеры определения и вызова функций

Вызов

- Оператор `def` создает функцию, но не вызывает ее.
- После выполнения `def` функцию можно вызывать (выполнять) в своей программе, добавляя к имени функции круглые скобки.
- Круглые скобки могут дополнительно содержать один и более объектов-аргументов, подлежащих передаче (присваиванию) именам в заголовке функции.

```
1 | >>> times(3, 5)
2 | 15
3 |
```

- Выражение вызова передает в `times` два аргумента.
- Аргументы передаются по порядку следования: имени `x` в заголовке функции присваивается значение 2, переменной `y` присваивается значение 4.
- Возвращаемый объект можно присвоить переменной:

```
1 | >>> x = times(3.14, 3)
2 | >>> x
3 | 9.42
4 |
```

Примеры определения и вызова функций

```
1 >>> def print_max(a, b):
2     ...     if a > b:
3     ...         print(a, "is max")
4     ...     elif a == b:
5     ...         print(a, "equals to", b)
6     ...     else:
7     ...         print(b, "is max")
8     ...
9
10 >>> print_max(6, 7)
11 7 is max
12
13 >>> print_max(3, 3)
14 3 equals to 3
15
16 >>> x, y = 5, 2
17
18 >>> print_max(x, y)
19 5 is max
20
```

- При объявлении переменных внутри определения функции, они никоим образом не связаны с другими переменными с таким же именем за пределами функции – т.е. имена переменных являются локальными в функции.
- Это называется **областью видимости переменной**. Область видимости всех переменных ограничена блоком, в котором они объявлены, начиная с точки объявления имени.

```
1 >>> x = 50
2
3 >>> def func(x):
4 ...     print("x =", x)
5 ...     x = 2
6 ...     print("Replace x to", x)
7 ...
8
9 >>> func(x)
10 x = 50
11 Replace x to 2
12
13 >>> print("x =", x)
14 x = 50
15
```

Оператор `global`

- Чтобы присвоить значение переменной, определённой на высшем уровне программы, необходимо явно указать Python, что её имя не локально, а глобально.
- Без применения зарезервированного слова `global` невозможно присвоить значение переменной, определённой за пределами функции.

```
1  >>> x = 50
2
3  >>> def func():
4  >>>     global x
5  ...
6  ...     print("x =", x)
7  ...     x = 2
8  ...     print("Replace x to", x)
9  ...
10
11 >>> func()
12 x = 50
13 Replace x to 2
14
15 >>> print("x =", x)
16 x = 2
17
```

Оператор `global`

```
1 x = 24 #Глобальная переменная x
2
3 def func():
4     x = 100 #Локальная переменная x
5
6 func()
7 print(x) #Выводит 100
8
```

```
1 y, z = 1, 2 #Глобальные переменные
2           #в модуле
3
4 def all_global():
5     global x #Объявление присваиваемой
6             #глобальной переменной
7     x = y + z #Объявлять y и z не нужно
8
```

- Добавленное объявление `global` привело к тому, что имя `x` внутри `def` теперь ссылается на имя `x` снаружи `def`.
- На этот раз они представляют собой одну и ту же переменную, а потому изменение `x` внутри функции приводит к изменению `x` за ее пределами.
- Переменные `y` и `z` Python ищет в модуле автоматически.
- Кроме того, до выполнения функции `all_global` переменная `x` вообще не существует во включающем модуле; в данном случае первое присваивание внутри функции создает переменную `x` в модуле.

Глобальные переменные только для крайних случаев

- Присваиваемые внутри `def` переменные по умолчанию будут локальными, поскольку так предусмотрено наилучшей стратегией.
- Изменение глобальных переменных может привести к проблемам при разработке программного обеспечения: из-за того, что значения переменных зависят от порядка вызовов произвольно отдаленных функций, программы могут стать трудными для отладки и для восприятия:

```
1 | x = 24
2 |
3 | def func1():
4 |     global x
5 |     x = 77
6 |
7 | def func2():
8 |     global x
9 |     x = 99
10 |
```

Для понимания кода понадобится отследить поток управления через целую программу. Вы не сможете использовать одну из функций без привлечения другой. Они зависят от глобальной переменной – т.е. связаны с ней. Если нужно повторно применить или модифицировать код – придется держать в уме всю программу целиком.

Области видимости и вложенные функции

Рассмотрим пример вложенной области видимости:

```

1  x = 24  #Имя в глобальной области видимости
2
3  def func1():
4      x = 55  #Локальное имя объемлющего def
5
6          def func2():
7              print(x)  #Ссылка во вложенном def
8
9              func2()
10
11 func1()  #Выводит 55: локальное имя объемлющего def
12
    
```

- Здесь вложенный оператор `def` запускается, пока выполняется функция `func1`; он создает объект функции и присваивает его имени `func2`, т.е. локальной переменной внутри локальной области видимости `func1`.
- В определенном смысле `func2` представляет собой временную функцию, которая существует только в период выполнения (и видима только в коде) объемлющей функции `func1`.

Области видимости и вложенные функции

- Поиск в объемлющей области видимости работает, даже если уже произошел возврат из объемлющей функции.
- В следующем коде определена функция, которая создает и *возвращает* объект другой функции, представляя более распространенный шаблон использования:

```
1 def func1():
2     x = 44
3
4 def func2():
5     print(x) #Помнит значение x из области видимости
6             #объемлющего def
7     return func2 #Возвращает объект функции func2, но не вызывает ее
8
9 action = f1() #Создает и возвращает объект функции
10 action() #Вызов функции: выводит 44
11
```

- Вызов `action` функцию `func2`, которая была создана во время выполнения `func1`.
- Функции в Python являются могут передаваться как возвращаемые значения.
- Функция `func2` помнит значение `x` из объемлющей области видимости функции `func1`, хотя `func1` больше неактивна.

- **Замыкание** – это методика *функционального программирования*, идея которой заключается в запоминании значений из объемлющих областей видимости, невзирая на то, присутствуют ли еще эти области видимости в памяти.
- Замыкания иногда применяются в программах, которым необходимо генерировать обработчики событий на лету в ответ на условия, сложившиеся во время выполнения.

```
1 >>> def maker(n):
2     ...     def action(x):           #Создание и возвращение функции action
3     ...         return x ** n       #action сохраняет n из объемлющей области
4     ...         #видимости
5     ...     return action
6
```

- В коде определяется внешняя функция, которая генерирует и возвращает вложенную функцию, не вызывая ее – `maker` создает `action`, и возвращает `action` без выполнения.
- Если вызвать внешнюю функцию – будет получена ссылка на вложенную функцию:

```
1 >>> f = maker(3)
2
3 >>> f
4 <function maker.<locals>.action at 0x000001987C301160>
5
```

- Вызов результата, возвращенного внешней функцией приводит к запуску вложенной функции, названной `action` внутри `maker`:

```
1 >>> f(5) #Перелача 5 аргументу x, в n запоминается 3: 5 ** 3
2 125
3 >>> f(2) #2 ** 3
4 8
5
```

- Если снова вызвать внешнюю функцию, то получим обратно новую вложенную функцию с другой информацией о состоянии:

```
1 >>> g = maker(2)
2
3 >>> g(5)
4 25
5
6 >>> f(5)
7 125
8
```

- Каждый вызов замыкания получает собственный набор информации о состоянии. Функция `g` запоминает 2, а `f` запоминает 3, т.к. каждая имеет свою информацию о состоянии, хранимую в переменной `n` из `maker`.

Оператор `nonlocal`

- Оператор `nonlocal` делает возможным присваивание значений именам из областей видимости объемлющих функций и ограничивает поиск таких имен областями видимости объемлющих `def`.
- Совокупным эффектом является более прямая и надежная реализация изменяемой информации о состоянии.

```
1  >>> def tester(start):
2  ...     state = start
3  ...     def nested(label):
4  ...         print(label, state)
5  ...         return nested
6  ...
7  >>> f = tester(1)
8  >>> f("hello")
9  hello 1
10 >>> f("hi ")
11 hi 1
12
```

- Функция `tester` создает и возвращает функцию `nested`, подлежащую вызову в более позднее время, а ссылка на `state` в `nested` отображается на имя в локальной области видимости `tester` с применением обычных правил поиска в областях видимости.

Оператор `nonlocal`

- Однако, изменение имени из области видимости объемлющего `def` по умолчанию не разрешено:

```
1  >>> def tester(start):
2  ...     state = start
3  ...     def nested(label):
4  ...         print(label, state)
5  ...         state += 1
6  ...     return nested
7  ...
8  >>> f = tester(2)
9  >>> f("hello")
10 Traceback (most recent call last):
11 File "<stdin>", line 1, in <module>
12 File "<stdin>", line 4, in nested
13 UnboundLocalError: local variable "state" referenced before assignment
14
```

Оператор `nonlocal`

- Если теперь объявить переменную `state` из области видимости `tester` как `nonlocal` внутри `nested`, то мы сможем ее также изменять во вложенной функции.
- Прием работает, несмотря на то, что к моменту вызова возвращенной функции `nested` через имя `f` функция `tester` уже завершилась.

```
1  >>> def tester(start):
2  ...     state = start
3  ...     def nested(label):
4  ...         nonlocal state
5  ...         print(label, state)
6  ...         state += 1
7  ...     return nested
8  ...
9  >>> f = tester(3)
10 >>> f("hello")
11 hello 3
12 >>> f("hi ")
13 hi 4
14 >>> f("good day")
15 good day 5
16
```

Строки документации (docstring)

- Документирование кода в python – важный аспект, т.к. от нее порой зависит читаемость и быстрота понимания Вашего кода, как другими людьми, так и Вами через некоторое время.
- Строки документации функции должны обобщить ее поведение и описать аргументы, возвращаемые значения, побочные эффекты и ограничения на вызов функции.

```
1  >>> def print_max(a, b):
2  ...     """
3  ...     Prints max value of a and b.
4  ...     NOTE: prints equals if a = b.
5  ...     """
6  ...     if a > b:
7  ...         print(a, "is max")
8  ...     elif a == b:
9  ...         print(a, "equals to", b)
10 ...     else:
11 ...         print(b, "is max")
12 ...
13 >>> print_max(3, 5)
14 5 is max
15 >>> help(print_max)
16 Help on function print_max in module __main__:
17 print_max(a, b)
18     Prints max value of a and b.
19     NOTE: prints equals if a = b.
20
```

Синтаксис передачи аргументов

Формы передачи аргументов функции

Формы передачи аргументов подразделяются на вызовы и определения функций.

Синтаксис	Местоположение	Интерпретация
<code>func(значение)</code> <code>func(имя=значение)</code> <code>func(*итерируемый_объект)</code>	Вызывающий код Вызывающий код Вызывающий код	Позиционный аргумент: передается по позиции Ключевой (именованный) аргумент: передается по имени Передает все объекты в итерируемом_объекте как отдельные позиционные аргументы
<code>func(**словарь)</code>	Вызывающий код	Передает все пары ключ/значение в словаре как отдельные именованные аргументы
<code>def func(имя)</code>	Функция	Позиционный аргумент: сопоставляется с любым переданным значением по позиции или по имени
<code>def func(имя=значение)</code>	Функция	Стандартное значение аргумента, если значение в вызове не передавалось
<code>def func(*имя)</code>	Функция	Передает и собирает оставшиеся позиционные аргументы в кортеж
<code>def func(**имя)</code>	Функция	Передает и собирает оставшиеся ключевые аргументы в словарь
<code>def func (*остальные, имя)</code>	Функция	Аргументы, которые должны передаваться в вызовах только по ключевому слову
<code>def func (*, имя=значение)</code>	Функция	Аргументы, которые должны передаваться в вызовах только по ключевому слову

- Если не использовать какой-то специальный синтаксис сопоставления, то Python будет сопоставлять имена по позиции слева направо подобно большинству других языков. Например, если Вы определили функцию, которая требует трех аргументов, тогда должны вызывать ее с тремя аргументами:

```
1  >>> def f(x, y, z):
2  ...     return x, y, z
3  ...
4
5  >>> f(0, 1, 2)
6  (0, 1, 2)
7
```

- Здесь аргументы передаются по позиции – x соответствует 0, y – 1 и z – 2.

Именованные параметры

- **Ключевые** (*именованные*) аргументы делают возможным сопоставление по имени, а не по позиции.

```
1 >>> f(z=2, x=0, y=1)
2 (0, 1, 2)
3
```

- Здесь z=2 означает передачу значения 2 аргументу по имени z.
- Когда применяются ключевые слова, порядок следования аргументов слева направо несущественен, поскольку аргументы сопоставляются по имени, а не по позиции.
- В одном вызове разрешено даже комбинировать позиционные и ключевые аргументы. Сначала сопоставляются все позиционные аргументы слева направо в заголовке, а затем ключевые аргументы сопоставляются по имени:

```
1 >>> f(0, z=2, y=1)
2 (0, 1, 2)
3
```

- Ключевые аргументы делают вызовы функций самодокументированными. Вызов следующего вида:

```
1 func(name="James", age=20, job="student")
2
```

выглядит более значащим по сравнению с вызовом, содержащим три разделенных запятыми значения, особенно в крупных программах.

Значения по умолчанию

- Стандартные значения позволяют делать некоторые аргументы функции необязательными.
- Если значение для аргумента не было передано, то ему присваивается стандартное значение.

```

1  >>> def f(x, y=1, z=2):    #Аргумент x обязательный
2  ...     return x, y, z    #y и z необязательные
3  ...
4  
```

- При вызове такой функции обязательно нужно предоставить значение для x, либо по позиции, либо по имени; однако передача значений для y и z необязательна:

```

1  >>> f(0)
2  (0, 1, 2)
3
4  >>> f(x=0)
5  (0, 1, 2)
6  
```

- В случае передачи двух значений стандартное значение получит только аргумент z, а при передаче трех значений стандартные значения вообще не применяются:

```

1  >>> f(10, 20)
2  (10, 20, 2)
3
4  >>> f(10, 20, 30)
5  (10, 20, 30)
6  
```

Заголовки функций: сбор аргументов

- Расширения при передаче аргументов `*` и `**` предназначены для поддержки функций, которые принимают *любое количество* аргументов. Оба расширения могут использоваться как в определении функции, так и при ее вызове.
- Когда расширение `*` применяется в определении функции, оно обеспечивает сбор несопоставленных позиционных аргументов в кортеж:

```
1 | >>> def f(*args):  
2 |     ...     return args  
3 |     ...
```

- При вызове такой функции Python собирает все позиционные аргументы в новый кортеж и присваивает его переменной `args`. Поскольку это нормальный объект кортежа, он допускает индексацию, проход в цикле `for` и т.д.:

```
1 | >>> f()  
2 | ()  
3 | >>> f(10)  
4 | (10,)  
5 | >>> f(10, 20, 30)  
6 | (10, 20, 30)  
7 |
```

Заголовки функций: сбор аргументов

- Расширение `**` работает с ключевыми аргументами – оно собирает их в словарь, который затем можно обрабатывать с помощью инструментов для словарей:

```
1 >>> def f(**kwargs):
2     ...     return kwargs
3     ...
4 >>> f()
5 {}
6
7 >>> f(x=2, y=3)
8 {"x": 2, "y": 3}
9
```

- В заголовках функций можно комбинировать позиционные аргументы с расширениями `*` и `**`, чтобы реализовывать гибкие сигнатуры вызовов.

```
1 >>> def f(x, *args, **kwargs):
2     ...     return x, args, kwargs
3     ...
4 >>> f(1, 2, 3, a=10, b=20)
5 (1, (2, 3), {"a": 10, "b": 20})
6
```

- Подобный код может встречаться в функциях, которым необходимо поддерживать множество шаблонов вызова.

Вызовы функций: распаковка аргументов

- Синтаксис `*` в контексте вызова функции имеет смысл, противоположный его смыслу в определении функции – он распаковывает коллекцию аргументов, а не собирает ее.

```
1 | >>> def f(a, b, c, d):  
2 |     ...     return a, b, c, d  
3 |     ...  
4 | >>> args = (10, 20, 30, 40, )  
5 | >>> f(*args)  
6 | (10, 20, 30, 40)  
7 |
```

- Подобным образом синтаксис `**` в вызове функции распаковывает словарь пар ключ / значение в отдельные именованные аргументы.

```
1 | >>> kwargs = {"a": 10, "b": 20, "c": 30, "d": 40}  
2 | >>> f(**kwargs)  
3 | (10, 20, 30, 40)  
4 |
```

Вызовы функций: распаковка аргументов

- Разрешено комбинировать позиционные и ключевые аргументы очень гибкими способами:

```
1 >>> def f(a, b, c, d):
2     ...     return a, b, c, d
3     ...
4 >>> f(10, 20, **{"d": 40, "c": 30})
5 (10, 20, 30, 40)
6
7 >>> f(*(10, 20), **{"d": 40, "c": 30})
8 (10, 20, 30, 40)
9
10 >>> f(10, *(20, 30), **{"d": 40})
11 (10, 20, 30, 40)
12
13 >>> f(10, c=30, *(2, ), **{"d": 40})
14 (10, 2, 30, 40)
15
16 >>> f(10, *(20, 30), d=40)
17 (10, 20, 30, 40)
18
19 >>> f(10, *(20, ), c=30, **{"d": 40})
20 (10, 20, 30, 40)
21
```

Рекурсивные функции

- *Рекурсивные функции* – это функции, которые вызывают самих себя либо прямо, либо косвенно с целью организации цикла.
- Рекурсия – довольно сложная тема и ее относительно редко можно встретить в коде Python, отчасти из-за того, что процедурные операторы Python включают более простые циклические структуры.
- Рекурсия является альтернативой несложным циклам и итерациям, хотя не обязательно более простой или эффективной.

Суммирование с помощью рекурсии

- Рассмотрим специальную реализацию функции суммирования с применением рекурсии:

```
1  >>> def my_sum(arr):
2  ...     if not arr:
3  ...         return 0
4  ...     return arr[0] + my_sum(arr[1:])
5  ...
6  >>> my_sum([10, 20, 30, 40, 50])
7  150
8
```

- На каждом уровне функция `my_sum` рекурсивно вызывает саму себя, чтобы вычислить сумму остатка списка, которая позже добавляется к элементу в начале списка.
- Когда список становится пустым, рекурсивный цикл заканчивается и возвращается ноль. В случае использования рекурсии такого рода каждый открытый уровень вызова функции имеет собственную копию локальной области видимости функции в стеке вызовов времени выполнения – здесь это означает, что переменная `arr` на каждом уровне разная.

Суммирование с помощью рекурсии

- Попробуем добавить в функцию вывод `arr` и запустить ее снова, чтобы отследить текущий список на каждом уровне вызова:

```

1  >>> def my_sum(arr):
2  ...     print(arr)
3  ...     if not arr:
4  ...         return 0
5  ...     return arr[0] + my_sum(arr[1:])
6  ...
7  >>> my_sum([10, 20, 30, 40, 50])
8  [10, 20, 30, 40, 50]
9  [20, 30, 40, 50]
10 [30, 40, 50]
11 [40, 50]
12 [50]
13 []
14 150
15
    
```

- Суммируемый список на каждом уровне рекурсии становится все меньше, пока окончательно не опустеет – конец рекурсивного цикла.
- Сумма вычисляется при раскручивании рекурсивных вызовов по возврату.

Сравнение операторов цикла с рекурсией

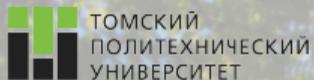
- В Python придается особое значение более простым процедурным операторам наподобие циклов, которые обычно намного естественнее.
- Например, `while` часто приносит чуть большую конкретику и не требует, чтобы функция определялась как допускающая рекурсивные вызовы:

```
1  >>> arr = [10, 20, 30, 40, 50]
2  >>> def sum_while(arr):
3  ...     s = 0
4  ...     while arr:
5  ...         s += arr[0]
6  ...         arr = arr[1:]
7  ...     return s
8  ...
9  >>> sum_while(arr)
10 150
11 >>> def sum_while(arr): #Альтернативный вариант
12 ...     s = 0
13 ...     while arr:
14 ...         s += arr.pop()
15 ...     return s
16 ...
17 >>> sum_while(arr)
18 150
19
```

Сравнение операторов цикла с рекурсией

- В дополнение циклы `for` обеспечивают автоматическую итерацию, делая рекурсию во многих случаях излишней и с высокой долей вероятности менее эффективной в плане расхода памяти и времени выполнения:

```
1  >>> arr = [1, 2, 3, 4, 5]
2  >>> def sum_while(arr):
3  ...     s = 0
4  ...     for x in arr:
5  ...         s += x
6  ...     return s
7  ...
8  >>> sum_while(arr)
9  150
10
```



Контакты

Вячеслав Алексеевич Чузлов,
к.т.н., доцент ОХИ ИШПР



Учебный корпус №2, ауд. 136



chuva@tpu.ru



+7-962-782-66-15

Благодарю за внимание!