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.

```
>>> def say_hi():
...     print("Hi!")

>>> say_hi()
Hi!
```

Оператор def

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

```
def имя_функции(аргумент1, аргумент2, ..., аргументN):
    оператор1  # операторы могут отсутствовать
    оператор2
    ...
    операторN
    return  # может быть пропущен
```

- 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 `return`

- Тело функции почти всегда содержит оператор `return`:

```
def имя_функции(аргумент1, аргумент2, ..., аргумент3):
    операторы
    ...
    return ...
```

- 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 возврата 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.

```
>>> def maximum(x, y):
...     if x > y:
...         return x
...
...     elif x == y:
...         return 'Equals.'
...
...     else:
...         return y

>>> print(maximum(2, 3))
3
```

```
>>> def maximum(x, y):
...     if x > y:
...         return x
...
...     if x == y:
...         return 'Equals.'
...
...     return y

>>> print(maximum(2, 3))
3
```

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

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

```
>>> def times(x, y):
...     return x * y
```

- Когда интерпретатор встречает и выполняет этот оператор `def`, он создает новый объект функции, умещающий 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

- Оператор `def` создает функцию, но не вызывает ее.
- После выполнения `def` функцию можно **вызывать** (выполнить) 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.

```
>>> times(3, 5)
15
```

- Выражение вызова передает в `times` два аргумента.
- Аргументы передаются по порядку следования: имени x в заголовке функции присваивается значение 3, переменной y присваивается значение 5.
- Возвращаемый объект можно присвоить переменной:

```
>>> x = times(3.14, 3)

>>> x
9.42
```

Примеры определения и вызова функций

```
>>> def print_max(a, b):
...     if a > b:
...         print(a, "is max")
...     elif a == b:
...         print(a, "equals to", b)
...     else:
...         print(b, "is max")
...

>>> print_max(6, 7)
7 is max

>>> print_max(3, 3)
3 equals to 3

>>> x, y = 5, 2

>>> print_max(x, y)
5 is max
```

Локальные переменные

- 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.

```
>>> x = 50

>>> def func(x):
...     print('x =', x)
...     x = 2
...     print('Replace x to', x)
...

>>> func(x)
x = 50
Replace x to 2

>>> print('x =', x)
x = 50
```

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.

```
>>> x = 50

>>> def func():
...     global x
...     print('x =', x)
...     x = 2
...     print('Replace x to', x)
...

>>> func()
x = 50
Replace x to 2

>>> print('x =', x)
x = 2
```

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: из-за того, что значения переменных зависят от порядка вызовов произвольно отдаленных функций, программы могут стать трудными для отладки и для восприятия:

```
x = 24

def func1():
    global x
    x = 77

def func2():
    global x
    x = 99
```

Для понимания кода понадобится отследить поток управления через целую программу. Вы не сможете использовать одну из функций без привлечения другой. Если нужно повторно применить или модифицировать код – придется держать в уме всю программу целиком.

Области видимости и вложенные функции

Рассмотрим пример вложенной области видимости:

```
>>> x = 24  # Имя в глобальной области видимости

>>> def func1():
...     x = 55  # Локальное имя объемлющего def
...
...     def func2():
...         print(x)  # Ссылка во вложенном def
...
...     func2()

>>> func1()  # Выводит 55: локальное имя объемлющего def
55
```

- 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:

```
>>> def func1():
...     x = 44
...
...     def func2():
...         print(x)  # Помнит значение x из области видимости объемлющего def
...
...     return func2  # Возвращает объект функции func2, но не вызывает ее

>>> action = func1()  # Создает и возвращает объект функции

>>> action()  # Вызов функции: выводит 44
44
```

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

Замыкания

- **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.

```
>>> def maker(n):
...     def action(x):      # Создание и возврат функции action
...         return x ** n   # action сохраняет значение n
...     return action
```

- 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:

```
>>> f = maker(3)

>>> f
<function maker.<locals>.action at 0x000001987C301160>
```

Замыкания

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

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

>>> f(2)  # 2 ** 3
8
```

- Если снова вызвать внешнюю функцию, то получим новую вложенную функцию с другой информацией о состоянии. Каждый вызов замыкания получает собственный набор информации о состоянии. Функция `g` запоминает 2, а `f` запоминает 3:

```
>>> g = maker(2)

>>> g(5)
25

>>> f(5)
125
```

Оператор `nonlocal`

- Оператор `nonlocal` разрешает присваивание значений именам из областей видимости объемлющих функций и ограничивает поиск таких имен этими областями.

```
>>> def tester(start):
...     state = start
...     def nested(label):
...         print(label, state)
...     return nested
...

>>> f = tester(1)

>>> f('hello')
hello 1

>>> f('hi')
hi 1
```

- 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:

```
>>> def tester(start):
...     state = start
...     def nested(label):
...         print(label, state)
...         state += 1
...     return nested

>>> f = tester(2)

>>> f('hello')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in nested
UnboundLocalError: local variable 'state' referenced before assignment
```

Оператор `nonlocal`

- Если теперь объявить переменную `state` из области видимости `tester` как **`nonlocal`** внутри `nested`, то мы сможем ее также изменять во вложенной функции.

```
>>> def tester(start):
...     state = start
...     def nested(label):
...         nonlocal state
...         print(label, state)
...         state += 1
...     return nested

>>> f = tester(3)

>>> f('hello')
hello 3

>>> f('hi')
hi 4

>>> f('good day')
good day 5
```

Строки документации (docstring)

- Строки документации (в тройных кавычках) должны обобщить поведение функции, описать аргументы, возвращаемые значения, побочные эффекты и ограничения на вызов функции.

```
>>> def print_max(a, b):
...     """
...     Prints max value of a and b.
...     NOTE: prints equals if a = b.
...     """
...     if a > b:
...         print(a, 'is max')
...     elif a == b:
...         print(a, 'equals to', b)
...     else:
...         print(b, 'is max')

>>> print_max(3, 5)
5 is max

>>> help(print_max)
Help on function print_max in module __main__:
print_max(a, b)
    Prints max value of a and b.
    NOTE: prints equals if a = b.
```

Формы передачи аргументов функции

Формы передачи аргументов подразделяются на вызовы и определения функций.

Синтаксис	Местоположение	Интерпретация
`func(значение)`	Вызывающий код	Позиционный аргумент: передается по позиции
`func(имя=значение)`	Вызывающий код	Ключевой (именованный) аргумент: передается по имени
`func(*итерируемый_объект)`	Вызывающий код	Передает все объекты в итерируемом_объекте как отдельные позиционные аргументы
`func(**словарь)`	Вызывающий код	Передает все пары ключ/значение в словаре как отдельные именованные аргументы
`def func(имя)`	Функция	Позиционный аргумент: сопоставляется с любым переданным значением по позиции или по имени
`def func(имя=значение)`	Функция	Стандартное значение аргумента, если значение в вызове не передавалось
`def func(*имя)`	Функция	Передает и собирает оставшиеся позиционные аргументы в кортеж
`def func(**имя)`	Функция	Передает и собирает оставшиеся ключевые аргументы в словарь
`def func(*остальные, имя1, имя2, ... имяN)`	Функция	Неограниченная передача позиционных аргументов; обязательная передача всех именованных аргументов по ключевому слову
`def func(*, имя1, имя2, ... имяN)`	Функция	Запрет позиционных аргументов; обязательная передача всех именованных аргументов по ключевому слову

Позиционные аргументы

- 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:

```
>>> def f(x, y, z):
...     return x, y, z
...

>>> f(0, 1, 2)
(0, 1, 2)
```

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

Именованные параметры

- **Именованные** аргументы делают возможным сопоставление по имени, а не по позиции.

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

- Здесь z=2 означает передачу значения 2 аргументу по имени z. Когда применяются ключевые слова, порядок следования аргументов несущественен, т.к. они сопоставляются по имени.
- Разрешено комбинировать позиционные и ключевые аргументы. Сначала сопоставляются позиционные аргументы слева направо в заголовке, а затем ключевые аргументы по имени:

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

- Ключевые аргументы делают вызовы функций самодокументированными. Вызов функции:

```
func(name='James', age=20, job='student')
```

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 используется стандартное значение.

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

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

```
>>> f(0)
(0, 1, 2)

>>> f(x=0)
(0, 1, 2)
```

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

```
>>> f(10, 20)
(10, 20, 2)

>>> f(10, 20, 30)
(10, 20, 30)
```

Заголовки функций: сбор аргументов

- Расширения при передаче аргументов `*` и `**` предназначены для поддержки функций, принимающих **любое количество** аргументов.
- Когда расширение `*` применяется в определении функции, оно обеспечивает сбор неограниченного количества позиционных аргументов в кортеж:

```
>>> def f(*args):
...     return args
...
```

- Python собирает все позиционные аргументы в кортеж и присваивает его переменной args. Поскольку это кортеж, допускается индексация, проход в цикле `for` и т.д.:

```
>>> f()
()

>>> f(10)
(10,)

>>> f(10, 20, 30)
(10, 20, 30)
```

Заголовки функций: сбор аргументов

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

```
>>> def f(**kwargs):
...     return kwargs
...
>>> f()
{}

>>> f(x=2, y=3)
{'x': 2, 'y': 3}
```

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

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

- 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.

```
>>> def f(a, b, c, d):
...     return a, b, c, d
...
>>> args = (10, 20, 30, 40, )

>>> f(*args)
(10, 20, 30, 40)
```

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

```
>>> kwargs = {'a': 10, 'b': 20, 'c': 30, 'd': 40}

>>> f(**kwargs)
(10, 20, 30, 40)
```

Вызовы функций: распаковка аргументов

- Разрешено комбинировать позиционные и ключевые аргументы очень гибкими способами:

```
>>> def f(a, b, c, d):
...     return a, b, c, d
...
>>> f(10, 20, **{'d': 40, 'c': 30})
(10, 20, 30, 40)

>>> f(*(10, 20), **{'d: 40, 'c': 30})
(10, 20, 30, 40)

>>> f(10, *(20, 30), **{'d': 40})
(10, 20, 30, 40)

>>> f(10, c=30, *(2, ), **{'d': 40})
(10, 2, 30, 40)

>>> f(10,  *(20, 30), d=40)
(10, 20, 30, 40)

>>> f(10,  *(20, ), c=30, **{'d': 40})
(10, 20, 30, 40)
```

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:

```
>>> def my_sum(arr):
...     if not arr:
...         return 0
...     return arr[0] + my_sum(arr[1:])
...
>>> my_sum([10, 20, 30, 40, 50])
150
```

- 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:

```
>>> def my_sum(arr):
...     print(arr)
...     if not arr:
...         return 0
...     return arr[0] + my_sum(arr[1:])
...
>>> my_sum([10, 20, 30, 40, 50])
[10, 20, 30, 40, 50]
[20, 30, 40, 50]
[30, 40, 50]
[40, 50]
[50]
[]
150
```

- 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 большинстве случаев рекурсивные функции избыточны.
- Например, цикл `while` часто привносит чуть большую конкретику и не требует рекурсий:

```
>>> arr = [10, 20, 30, 40, 50]

>>> def sum_while(arr):
...     s = 0
...     while arr:
...         s += arr[0]
...         arr = arr[1:]
...     return s
...
>>> sum_while(arr)
150

>>> def sum_while(arr):  # Альтернативный вариант
...     s = 0
...     while arr:
...         s += arr.pop()
...     return s
...
>>> sum_while(arr)
150
```

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:

```
>>> arr = [1, 2, 3, 4, 5]

>>> def sum_for(arr):
...     s = 0
...     for x in arr:
...         s += x
...     return s
...
>>> sum_for(arr)
150
```