Сообщить об ошибке.

Все особенности определения функции в Python

Функции в Python определяются с помощью инструкции def, которое вводит определение функции. За ним должно следовать имя функции и заключенный в скобки список формальных параметров/аргументов. Операторы, которые формируют тело функции, начинаются со следующей строки и должны иметь отступ.

```
def func_name(param):
    pass
```

- func_name - идентификатор, то есть переменная, которая при выполнении инструкции def связывается со значением в виде объекта функции.
- param - это необязательный список формальных параметров/аргументов, которые связываются со значениями, предоставляемыми при вызове функции. В простейшем случае функция может вообще не иметь параметров/аргументов. Это означает, что при ее вызове она не получает никаких аргументов. В определении такой функций круглые скобки после ее имени остаются пустыми, такими же они должны оставаться при ее вызове.

Первым оператором тела функции может быть строка документации функции. Существуют инструменты, которые используют строки документации для автоматического создания интерактивной или печатной документации или для предоставления пользователю интерактивного просмотра кода. Хорошей практикой является включение строк документации в код.

Определение функции - это исполняемый оператор. Его выполнение связывает имя функции в текущем локальном пространстве имен с объектом функции (оболочка вокруг исполняемого кода функции). Этот объект функции содержит ссылку на текущее глобальное пространство имен, которое будет использоваться при вызове функции.

> *Определение функции не выполняет тело функции, а только вычисляет аргументы, если они присутствуют. Тело функции выполняется только при вызове.*

```
# Определим функцию, которая печатает список чисел ряда Фибоначчи до n
>>> def fib(n):
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Вызов функции
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

При вызове функции создается область видимости для хранения локальных переменных этой функции и все созданные переменные внутри функции сохраняют свое значение только в этом пространство имен. Когда функция завершает выполнение, это пространство имен уничтожается.

Определение функции может быть заключено в одно или несколько выражений-декоратора. Выражения-декораторы оцениваются при определении функции в области, содержащей определение функции. Результатом должен быть вызываемый объект, который вызывается с объектом функции в качестве единственного аргумента. Возвращаемое значение привязывается к имени функции, а не к объекту функции. Несколько декораторов применяются вложенным образом. Например, следующий код

```
@f1(arg)
@f2
def func(): pass

# Примерно эквивалентен

def func(): pass
func = f1(arg)(f2(func))
```

Эти два выражения эквивалентны за исключением того, что исходная функция временно не привязана к имени func.

Когда один или несколько параметров имеют выражение parameter=expression, то говорят, что функция имеет значения аргумента по умолчанию. Для параметра со значением по умолчанию, соответствующий аргумент может быть опущен в вызове. Если параметр имеет значение по умолчанию, то все последующие параметры/аргументы также должны иметь значение по умолчанию - это синтаксическое ограничение.

Значения параметров по умолчанию оцениваются слева направо при определении функции. Это означает, что эти выражение вычисляется один раз, когда функция определена, и что для каждого вызова используется одно и то же предварительно вычисленное значение. Это особенно важно понимать, когда параметр по умолчанию является изменяемым объектом, таким как список или словарь

Семантика вызова функций более подробно описана в разделе "Что происходит в момент вызова функции?". Вызов функции всегда присваивает значения всем параметрам, упомянутым в списке параметров, либо из позиционных аргументов, из ключевых аргументов, либо из значений по умолчанию. Если присутствует форма *identifier, то она инициализируется кортежем, которая получает любые избыточные позиционные параметры, по умолчанию - пустой кортеж. Если присутствует форма **identifier, то она инициализируется новым упорядоченным словарем, получающим любые избыточные ключевые аргументы, по умолчанию используется новый пустой словарь.

Параметры, следующие после * или *identifier являются параметрами только ключевых слов и могут быть переданы только в качестве ключевых аргументов (более подробно в разделе "Варианты передачи аргументов в функцию Python").

Параметры могут иметь аннотацию типов в форме :expression после имени аргумента. Любой параметр может иметь аннотацию, даже в форме *identifier или **identifier. Функции могут иметь аннотацию возвращаемого значения в форме -> expression, которое следует после их списка. Эти аннотации могут быть любым допустимым выражением Python. Наличие аннотаций не меняет семантику функции. Значения аннотации доступны в виде значений словаря с ключами по именам параметров в атрибуте объекта функции __annotations__ и оцениваются при выполнении определения функции. Если используется импорт аннотаций из __future__, то аннотации сохраняются в виде строк во время выполнения, что позволяет отложить оценку. В этом случае аннотации могут оцениваться в другом порядке, чем они появляются в исходном коде.

Также возможно создавать анонимные функции (функции, не привязанные к имени) для немедленного использования в выражениях. Здесь используются лямбда-выражения, описанные в разделе "Анонимные функции (lambda-выражения) в Python". Обратите внимание, что лямбда-выражение - это просто сокращение для упрощенного определения функции. Функция, определенная в операторе def, может быть передана или присвоена другому имени, как функция, определенная лямбда-выражением. Форма def на самом деле более мощная, так как она позволяет выполнять несколько операторов и аннотаций.

Примечание для программиста: Функции Python - это объекты первого класса. Оператор def, выполняемый внутри определения функции, определяет локальную функцию, которая может быть возвращена или передана. Свободные переменные, используемые во вложенной функции, могут обращаться к локальным переменным функции, содержащей def. Подробнее смотрите в раздел Именование и привязка.

Так как функции в Python являются объектами первого класса, то значение имени функции имеет тип, который распознается интерпретатором как определяемая пользователем функция. Это значение может быть присвоено другому имени, которое затем может также использоваться как функция:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
>>>
```

Исходя из других языков, можно возразить, что fib это не функция, а процедура, поскольку она не возвращает значение. На самом деле, даже функции без оператора return возвращают значение, хотя и довольно скучное. Это значение называется None (встроенное имя). Вывод значения None обычно подавляется интерпретатором, если это единственное возвращаемое значение. Вы можете увидеть это, используя команду print():

```
>>> fib(0)
>>> print(fib(0))
None
>>>
```

Функция, которая возвращает список чисел ряда Фибоначчи, а не печатает его:

```
>>> def fib2(n):  # возврат ряда Фибоначчи до n
...     """Возвращает список, содержащий ряд Фибоначчи до n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # вызов функции
>>> f100
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>>
```

Этот пример, демонстрирует некоторые новые возможности Python:

- Оператор return возвращает значение из функции. return без аргумента возвращает None. Функции, у которых return не определен, также возвращает None.
- Оператор result.append(a) вызывает метод объекта списка result. Метод - это функция, которая «принадлежит» объекту и имеет имя obj.methodname, где obj есть некоторый объект (может быть выражение), и methodname имя метода, которое определяется типом объекта. Разные типы определяют разные методы. Методы разных типов могут иметь одно и то же имя, не вызывая двусмысленности. Можно определить свои собственные типы объектов и методы, используя классы. Метод append(), показанный в примере, определен для объектов списка. Он добавляет новый элемент в конец списка.

Содержание раздела:

- Рекурсия
- Замыкания в функциях Python
- Перегрузка функций