Лекция 6.1. Функции. Основные понятия и элементы

# **Введение**

Функции являются фундаментальной частью любого программного языка, и Python не исключение. Они позволяют разбивать код на логически обособленные блоки, что делает его более понятным, модульным и переиспользуемым. Функции в Python предоставляют возможность выполнять определенные действия или вычисления, вызывая их по имени, и могут принимать входные параметры и возвращать значения.

В этой статье мы рассмотрим, как определить и объявить функции в Python, включая основные принципы их создания и использования. Также мы обсудим перегрузку функций, которая позволяет использовать одно и то же имя для различных реализаций функции, и функции с переменным количеством аргументов, что делает функции более гибкими и универсальными. Понимание этих концепций поможет вам писать более эффективный и удобочитаемый код.

# **Функции. Объявление и определение функции. Формальные и фактические параметры. Способы передачи параметров. Вызов функции. Область действия имен.**

В этой части мы изучим функции — составные инструкции, которые могут принимать данные ввода, выполнять указания и возвращать данные вывода. Функции позволяют определять и повторно использовать определенную функциональность в компактной форме.

Вызвать функцию — значит передать ей входные данные, необходимые для выполнения и возвращения результата. Когда вы передаете функции входные данные, это называется передача параметра функции.

Функции в Python похожи на математические функции из алгебры. Например, в алгебре функция определяется так:



Левая часть определяет функцию f, принимающую один параметр, x. А правая часть — это определение функции, которое использует переданный параметр x, чтобы произвести вычисление и вернуть результат. В этом случае значением функции является ее параметр, умноженный на два.

Как в Python функция записывается следующим образом: имя\_функции(параметры\_через\_запятую). Чтобы вызвать функцию, после ее имени нужно указать круглые скобки и поместить внутрь параметры, отделив каждый из них запятой. Для создания функций в Python выберите ее имя, определите параметры, укажите, что функция должна делать и какое значение возвращать.



Математическая функция f(x) = x \* 2 в Python будет выглядеть вот так:



Ключевое слово def сообщает Python, что вы определяете функцию. После def вы указываете имя функции; оно должно отвечать тем же правилам, что и имена переменных. Согласно конвенции, в имени функции нельзя использовать заглавные буквы, а слова должны быть разделены подчеркиванием вот\_так.

Как только вы присвоили своей функции имя, укажите после него круглые скобки. Внутри скобок должен содержаться один или несколько параметров.

После скобок ставится двоеточие, а новая строка начинается с отступа в четыре пробела. Любой код с отступом в четыре пробела после двоеточия является телом функции. В этом случае тело нашей функции состоит только из одной строки:



Ключевое слово return используется для определения значения, которое функция возвращает при вызове.

Чтобы вызвать функцию в Python, мы используем синтаксис имя\_функции(параметры, через, запятую).

Ниже описан вызов функции f из предыдущего примера с параметром 2.



Консоль ничего не вывела. Можно сохранить вывод вашей функции в переменной и передать ее функции print.



Вы можете сохранить результат, возвращаемый вашей функцией, в переменной и использовать это значение в программе позднее.



У функции может быть один параметр, несколько параметров или вообще их не быть. Чтобы определить функцию, не требующую параметров, оставьте круглые скобки пустыми.



Если хотите, чтобы функция принимала больше одного параметра, отделите каждый параметр в скобках запятой.



Наконец, функция не обязана содержать инструкцию return. Если функции нечего возвращать, она возвращает значение None.



Обязательные и необязательные параметры

Функция может принимать параметры двух типов. Те, что встречались вам до этого, называются обязательными параметрами. Когда пользователь вызывает функцию, он должен передать в нее все обязательные параметры, иначе Python сгенерирует исключение.

В Python есть и другой вид параметров — опциональные. Опциональные параметры определяются с помощью следующего синтаксиса: имя\_функции(имя\_параметра = значение\_параметра). Как и обязательные, опциональные параметры нужно отделять запятыми. Ниже приведен пример функции, в коде которой используется опциональный параметр.



Сначала функция вызывается без передачи параметра. Так как параметр необязательный, x автоматически становится равен 2, и функция возвращает 4.

Затем та же функция вызывается с параметром 4. То есть x будет равен 4 и функция вернет 16. Вы можете определить функцию, которая принимает как обязательные, так и опциональные параметры, но обязательные нужно определять в первую очередь.



Функции Python — это объекты первого класса. Их можно присваивать переменным, хранить в структурах данных, передавать в качестве аргументов другим функциям и даже возвращать в качестве значений из других функций.

Интуитивное понимание этих понятий значительно облегчит понимание таких продвинутых функций Python, как лямбды и декораторы. А также поможет вам продвинуться на пути к техникам функционального программирования.

В этом руководстве я приведу ряд примеров, которые помогут развить это интуитивное понимание. Каждый последующий пример будет опираться на предыдущий, поэтому вам, возможно, захочется читать их последовательно и даже опробовать некоторые из них в сессии интерпретатора Python по ходу дела.

На то, чтобы разобраться в концепциях, которые мы будем здесь обсуждать, может потребоваться больше времени, чем вы ожидаете. Не волнуйтесь, это совершенно нормально. Со мной такое бывало. Вам может показаться, что вы безрезультатно бьетесь головой о стену, а потом в голове вдруг «щелкнет» и все встанет на свои места, когда вы будете к этому готовы.

В этом руководстве я буду использовать в демонстрационных целях функцию yell. Это простой пример с легко узнаваемым результатом:



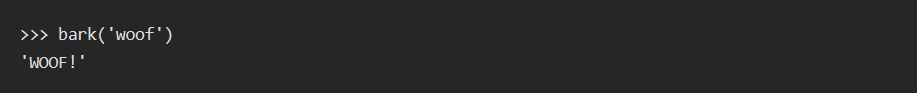
Функции — это объекты

Все данные в программе Python представлены объектами или отношениями между объектами. Строки, списки, модули и функции — все это объекты. В Python нет ничего особенного в функциях.

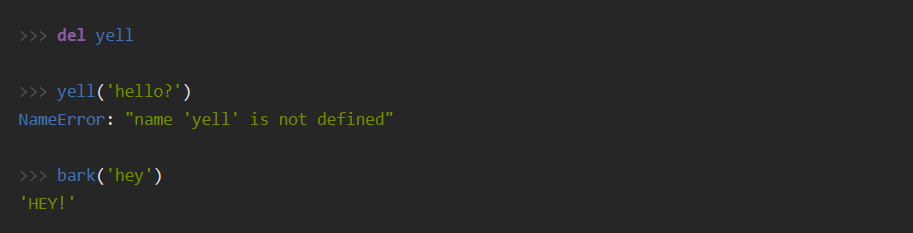
Поскольку функция yell является объектом в Python, вы можете присвоить ее другой переменной, как и любой другой объект:



Эта строка не вызывает функцию. Она берет объект функции, на который ссылается yell, и создает второе имя, указывающее на него, bark. Теперь можно выполнить тот же базовый объект функции, вызвав bark:



Объекты функций и их имена — это две разные вещи. Вот еще одно доказательство: исходное имя функции (yell) можно удалить. Поскольку другое имя (bark) все еще указывает на базовую функцию, вы все еще можете вызвать функцию через него:



Кстати, Python для целей отладки во время создания прикрепляет строковый идентификатор к каждой функции. Доступ к этому внутреннему идентификатору можно получить с помощью атрибута name:

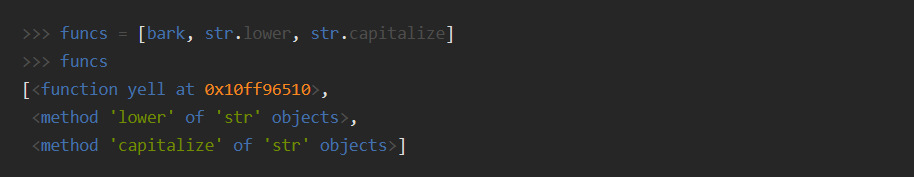


Хотя функции name по-прежнему "yell", это не влияет на то, как можно получить к ней доступ из кода. Этот идентификатор — всего лишь средство отладки. Переменная, указывающая на функцию, и сама функция — это две разные вещи.

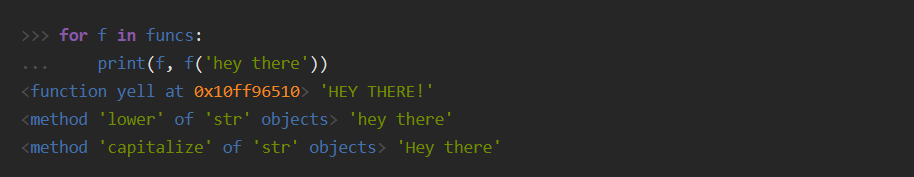
Начиная с Python 3.3 существует также qualname, который служит аналогичной цели и предоставляет строку квалифицированного имени для разграничения имен функций и классов.

Функции можно хранить в структурах данных

Функции можно хранить в структурах данных, как и другие объекты. Например, можно добавить функции в список:



Доступ к объектам функций, хранящихся внутри списка, осуществляется так же, как и к любому другому типу объектов:

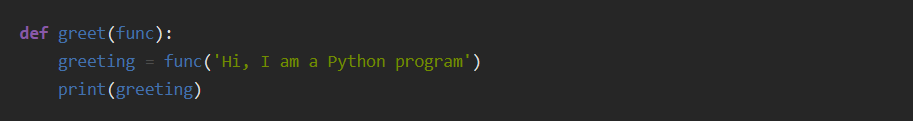


Можно даже вызвать объект функции, хранящийся в списке, не присваивая его сначала переменной. Можно выполнить поиск и затем сразу же вызвать полученный «свободный» объект функции в рамках одного выражения:



Функции можно передавать другим функциям

Поскольку функции являются объектами, их можно передавать в качестве аргументов другим функциям. Вот функция greet, которая форматирует строку greeting, используя переданный ей объект функции, а затем выводит ее:



Приветствие, которое получается в результате, можно изменить, передавая различные функции. Вот что произойдет, если передать в greet функцию yell:



Чтобы сгенерировать другой вид приветствия, можно определить новую функцию, Например, функция whisper может работать лучше, если вы не хотите, чтобы программы на Python звучали так, словно их автор Оптимус Прайм:

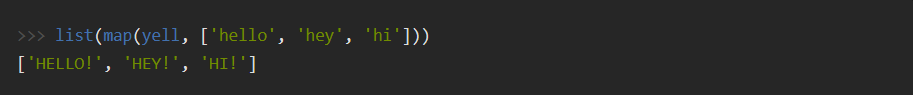


Мощной является возможность передавать объекты функций в качестве аргументов другим функциям. Она позволяет абстрагироваться от поведения и передавать его в программах. В этом примере функция greet остается неизменной, но на ее вывод можно влиять, передавая различные варианты приветствия.

Функции, которые могут принимать другие функции в качестве аргументов, также называются функциями высшего порядка. Они необходимы для функционального стиля программирования.

Классическим примером функций высшего порядка в Python является встроенная функция map. Она принимает функцию и итерируемый объект и вызывает функцию на каждом элементе в итерируемом объекте, выдавая результаты по мере выполнения.

Вот как можно отформатировать последовательность всех приветствий сразу, с помощью маппинга функции yell:



map просмотрел весь список и применил функцию yell к каждому элементу.

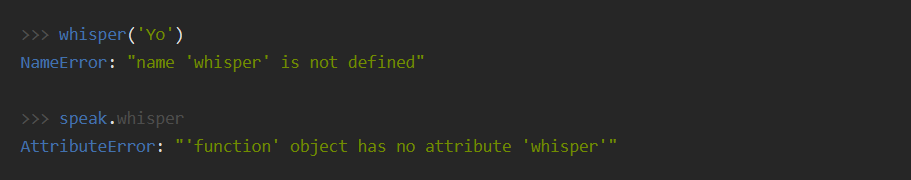
Функции могут быть вложенными

Python позволяет определять функции внутри других функций. Их часто называют вложенными или внутренними функциями. Вот пример:



Итак, что же здесь происходит? Каждый раз, когда вы вызываете speak, он определяет новую внутреннюю функцию whisper и затем вызывает ее.

И вот в чем загвоздка — whisper не существует вне speak:

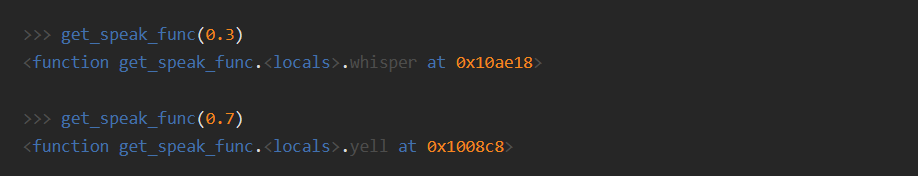


Но что, если вы действительно хотите получить доступ к вложенной функции whisper из внешней функции speak? Ну, функции являются объектами — можно вернуть внутреннюю функцию вызывающей родительской функции.

Например, вот функция, определяющая две внутренние функции. В зависимости от аргумента, переданного функции верхнего уровня, она выбирает и возвращает вызывающему одну из внутренних функций:



Обратите внимание, что get\_speak\_func фактически не вызывает ни одну из своих внутренних функций — она просто выбирает соответствующую функцию на основе аргумента volume, а затем возвращает объект функции:



Конечно, можно продолжить и вызвать возвращенную функцию либо напрямую, либо присвоив ее имя переменной:



Подумайте об этом немного. Это означает, что функции могут не только принимать поведение через аргументы, но и возвращать его. Круто же?

Знаете что, на этом моменте всё начинает лихо закручиваться. Прежде чем продолжить, я сделаю небольшой перерыв на кофе (и вам я советую сделать то же самое).

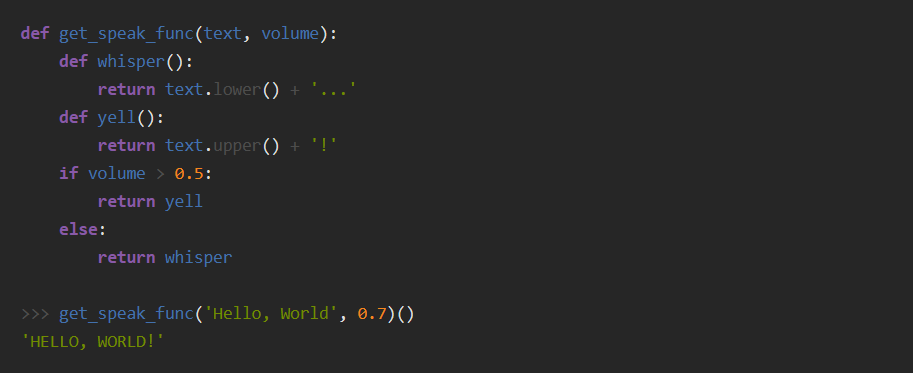
Функции могут захватывать локальное состояние

Вы только что увидели, что функции могут содержать внутренние функции и что можно даже возвращать эти (иначе скрытые) внутренние функции из родительской функции.

А теперь пристегнитесь, потому что дальше будет еще безумнее — мы собираемся погрузиться в еще более глубокую область функционального программирования. (У вас ведь был перерыв на кофе, верно?)

Функции не только могут возвращать другие функции. Эти внутренние функции также могут захватывать и переносить с собой некоторые состояния родительской функции.

Чтобы проиллюстрировать эту мысль, я немного перепишу предыдущий пример get\_speak\_func. Чтобы сделать возвращаемую функцию сразу вызываемой, новая версия сразу принимает аргументы «громкость» и «текст»:



Посмотрите внимательно на внутренние функции whisper и yell. Заметили, что у них больше нет параметра text? Но каким-то образом они все еще могут обращаться к параметру text, определенному в родительской функции. На самом деле, они, похоже, захватывают и «запоминают» значение этого аргумента.

Функции, которые делают это, называются лексическими замыканиями (или просто замыканиями, для краткости). Замыкание запоминает значения из своей лексической области видимости, даже когда поток программы больше не находится в этой области.

На практике это означает, что функции могут не только возвращать поведение, но и настраивать это поведение. Вот еще один пример, иллюстрирующий эту идею:



В этом примере make\_adder служит фабрикой для создания и настройки функций "adder". Обратите внимание, что функции "adder" по-прежнему могут обращаться к аргументу n функции make\_adder (вложенная область видимости).

Объекты могут вести себя как функции

Объекты не являются функциями в Python. Но их можно сделать вызываемыми, что позволяет во многих случаях обращаться с ними как с функциями.

Если объект является вызываемым, для него можно использовать круглые скобки () и передавать ему аргументы вызова функции. Вот пример вызываемого объекта:



За кулисами «вызов» экземпляра объекта как функции пытается выполнить метод call объекта.

Конечно, не все объекты могут быть вызываемыми. Поэтому существует встроенная функция callable для проверки того, является ли объект вызываемым или нет:



Основные выводы

* В Python все является объектом, включая функции. Их можно присваивать переменным, хранить в структурах данных, передавать или возвращать другим функциям (первоклассным функциям).
* Функции первого класса позволяют абстрагироваться и передавать поведение в программах.
* Функции могут быть вложенными, они могут захватывать и переносить с собой часть состояния родительской функции. Функции, которые делают это, называются замыканиями.
* Объекты можно сделать вызываемыми, что позволяет во многих случаях обращаться с ними как с функциями.

# **Перегрузка функций. Функции с переменным числом параметров.**

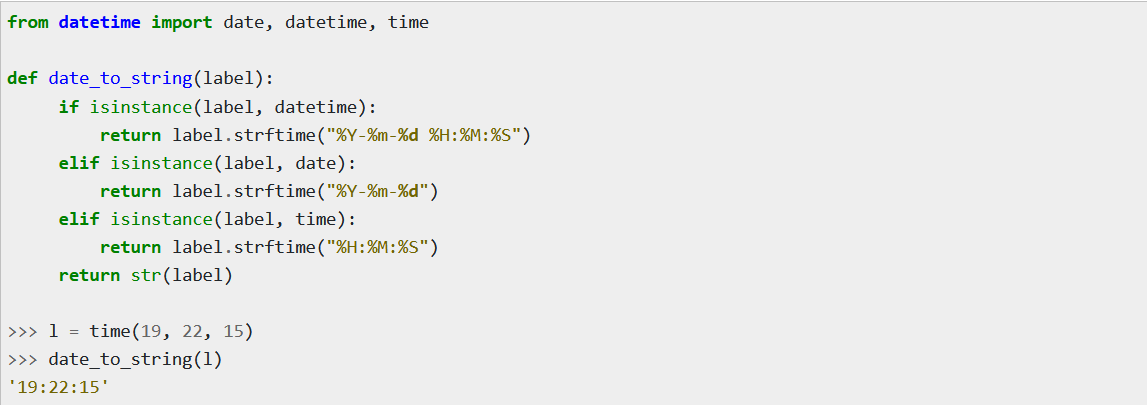
Перегрузка функций, по-другому универсальные функции — это распространенный шаблон программирования, называемый множественной диспетчеризацией/отправкой, который, наверное, зарезервирован только для статически типизированных, компилируемых языков. Python является динамически типизированным языком и следовательно, перегрузка функций здесь невозможна, тем не менее, есть простой способ реализовать такое поведение в Python.

И так, как же можно реализовать перегрузку функции в Python? Ведь для перегрузки функции требуется, чтобы язык программирования мог различать типы во время компиляции. Для этого, интерпретатору Python необходимо различить несколько реализаций функции во время выполнения, на основе динамически определенных типов. Другими словами, Python нужно как-то использовать типы аргументов, передаваемых функции во время ее вызова, и на их основе динамически выбирать, какую из нескольких реализаций функции использовать.

Кто не стакивался со строго типизированными языками может задаться вопросом: "А оно нам нужно? Если такое поведение не может быть реализовано нормально, то и не стоит его использовать...". Да, это так, но есть веские причины реализовать поведение перегрузки функций в Python. Это мощный инструмент, который может сделать код более кратким, читаемым и минимизировать его сложность.

Без поддержки множественной диспетчеризации (отправки) очевидный способ сделать это в Python - использовать проверку типов встроенной функцией isinstance(). Это очень хрупкое решение, которое закрыто для расширения кода, а некоторые программисты, пришедшие с типизированных языков, называют его анти-паттерном.

Например:

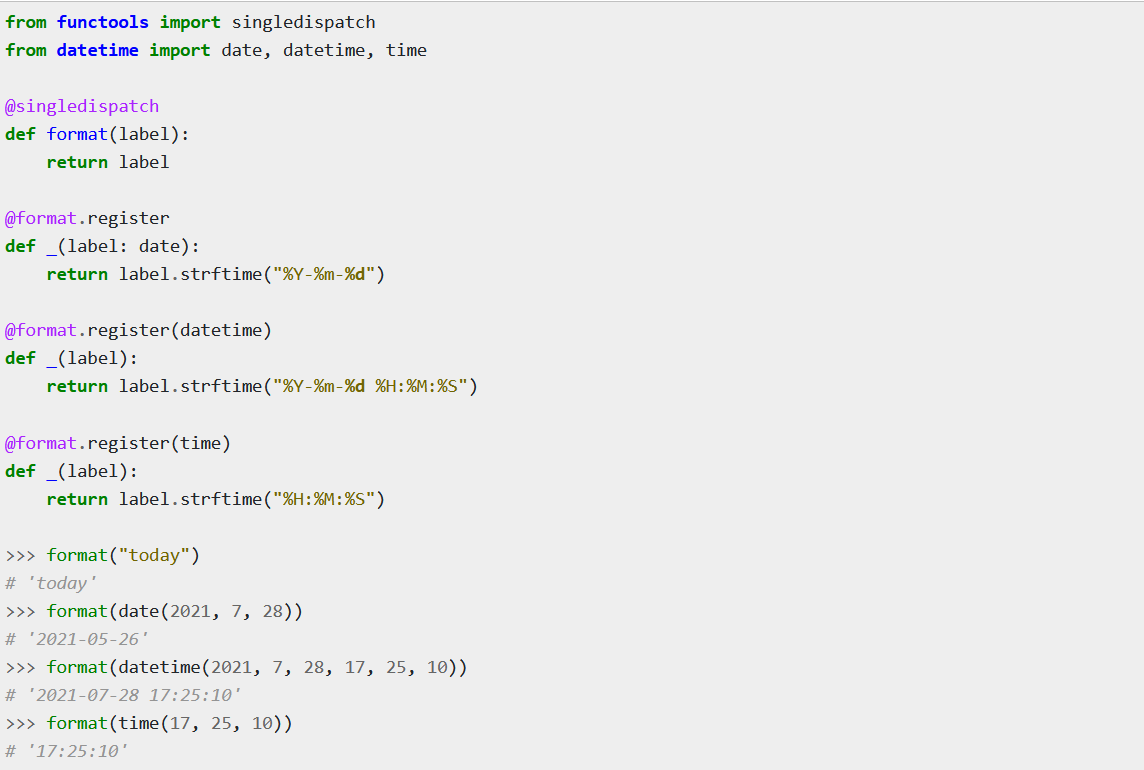


В примере, на основе типа, передаваемого аргумента label, принимается решение о строковом представлении даты и/или времени. Представьте что будет с кодом этой функции, если для получения результата определенного типа необходима длинная логика? Придется использовать фабрику функций, но проще и понятней использовать перегрузку.

Перегрузка одного аргумента (одиночная отправка).

В Python нет функции или класса стандартной библиотеки, которая поддерживает множественную отправку, но доступна одиночная отправка. Фактическое различие между множественной и одиночной отправкой - это количество аргументов, которые можно перегрузить. Функция (и декоратор), предоставляющая эту возможность, называется @singledispatch и расположена в модуле functools.

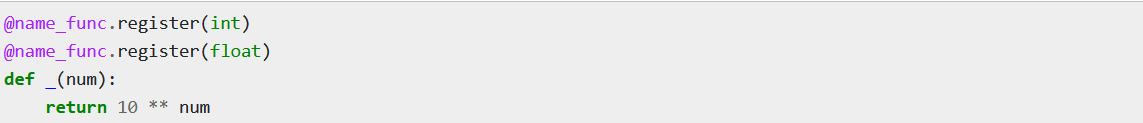
Эту концепцию лучше всего объяснить на нескольких примерах. Пример использования декоратора @singledispatch при форматировании datetime.datetime, datetime.date и datetime.time:



В данном коде, длинная логика получения значения для определенного типа аргумента не страшна, так как для каждого типа определяется отдельная функция, при этом основная функция своего названия не меняет.

В примере определяется базовая функция format(), которая будет перегружена. Эта функция декорируется @singledispatch и обеспечивает базовую реализацию, которая используется, если нет подходящего типа. Затем определяются отдельные дочерние функции для каждого типа, которые необходимо перегрузить - в данном случае это datetime.date, datetime.datetime и datetime.time - каждый из них имеет имя \_ (подчеркивание), т.к. все они будут вызваны (отправлены) через функцию format(). Каждая из дочерних функций также декорируется @format.register(), что прикрепляет их к ранее упомянутой функции format(). Чтобы дочерние функции различали типы, есть два варианта - можно использовать аннотации типов, как показано в первом случае, или явно передать тип в декоратор.

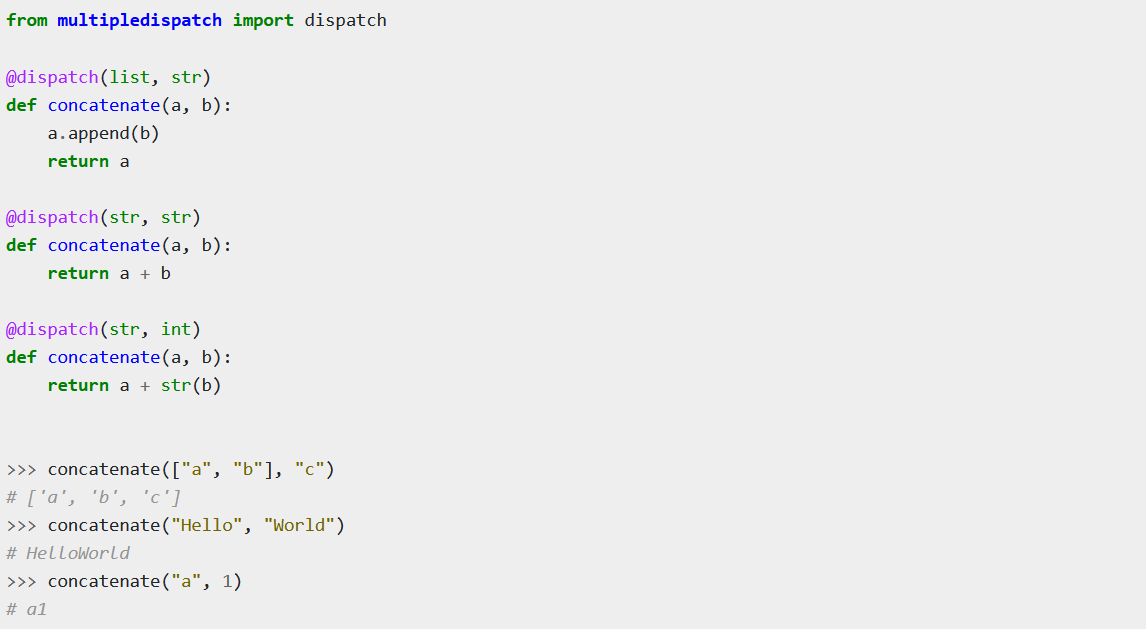
В некоторых случаях имеет смысл использовать одну и ту же реализацию для нескольких типов, например, для int и float. Для таких ситуаций допускается перечисление нескольких декораторов @format.register(type), чтобы связать функцию со всеми допустимыми типами.



Перегрузка нескольких аргументов (множественная отправка).

Часто одиночной отправки будет недостаточно, и может понадобиться надлежащая функциональность множественной отправки. Поведение множественной отправки доступно из стороннего модуля multipledispatch.

Модуль multipledispatch и его декоратор @dispatch ведут себя очень похоже на @singledispatch. Единственная разница заключается в том что он может принимать в качестве аргументов несколько типов:



Приведенный выше фрагмент показывает, как можно использовать декоратор @dispatch для перегрузки функции с несколькими аргументами, например для реализации конкатенации различных типов. Со сторонним модулем multipledispatch не нужно определять и регистрировать базовую функцию, а просто создать несколько функций с одним и тем же именем. Если необходимо предоставить базовую реализацию, то можно использовать @dispatch(object, object) для улавливания любыми неспецифическими типами аргументов.

## **\*args и \*\*kwargs**

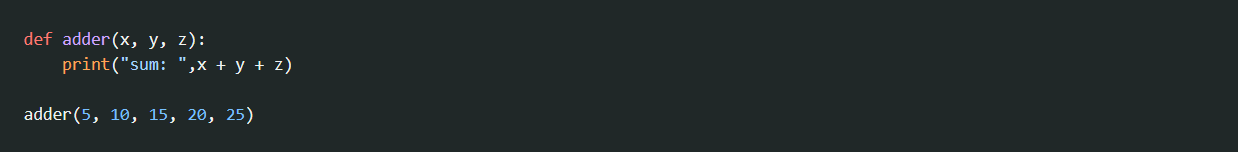
Не всегда заранее известно, сколько аргументов будет передано функции. В таких случаях в Python на помощь приходят \*args и \*\*kwargs, позволяющие передавать переменное количество аргументов. Сегодня мы расскажем, как именно они работают.



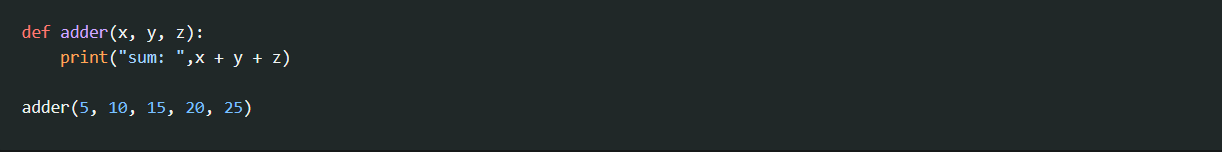
В этой лекции мы расскажем, зачем нужны  \*args и \*\*kwargs в Python и как их использовать.

В программировании, если нам нужно выполнять похожие действия, мы определяем функции для многоразового использования кода. Чтобы выполнить это действие, мы вызываем функцию с определённым значением — аргументом.

Предположим, у нас есть функция, которая складывает три числа:

После запуска будет выведено sum: 35.

Во фрагменте кода выше у нас есть функция adder() с тремя аргументами: x, y и z. При передаче трёх значений этой функции на выходе мы получаем их сумму. Но что, если передать больше трёх аргументов в эту функцию?

Из-за того, что здесь мы передаём 5 аргументов, при запуске программы выводится ошибка TypeError: adder() takes 3 positional arguments but 5 were given.

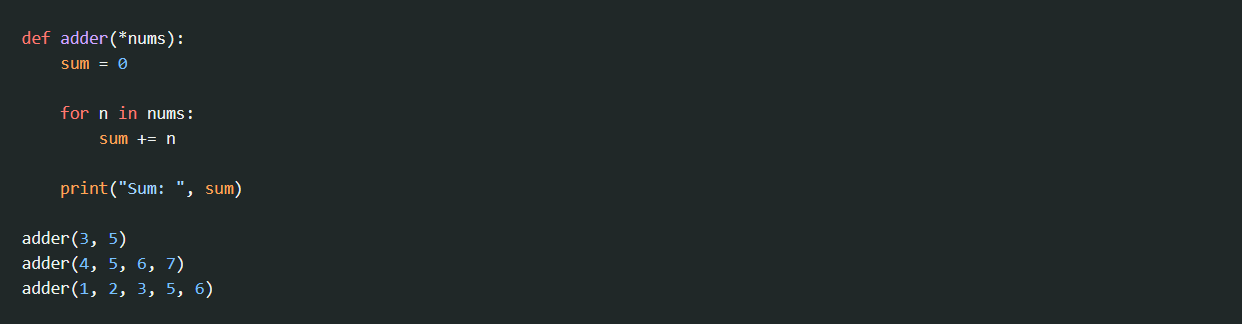
В Python можно передать переменное количество аргументов двумя способами:

* \*args для неименованных аргументов;
* \*\*kwargs для именованных аргументов.

Мы используем \*args и \*\*kwargs в качестве аргумента, когда заранее не известно, сколько значений мы хотим передать функции.

## **\*args**

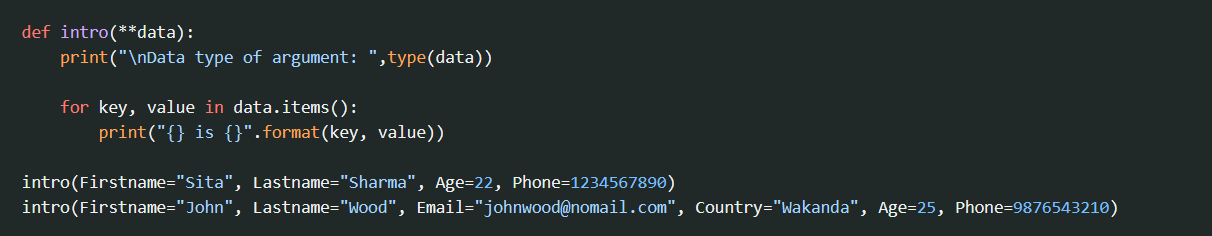
Как было сказано, \*args нужен, когда мы хотим передать неизвестное количество неименованных аргументов. Если поставить \* перед именем, это имя будет принимать не один аргумент, а несколько. Аргументы передаются как кортеж и доступны внутри функции под тем же именем, что и имя параметра, только без \*. Например:

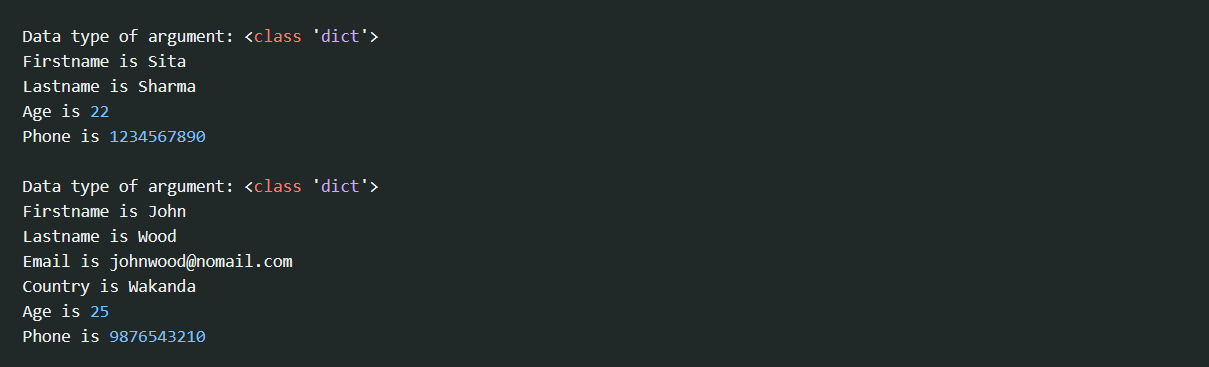
В результате выполнения программы мы получим следующий результат:

Здесь мы использовали \*nums в качестве параметра, который позволяет передавать переменное количество аргументов в функцию adder(). Внутри функции мы проходимся в цикле по этим аргументам, чтобы найти их сумму, и выводим результат.

## **\*\*kwargs**

По аналогии с \*args мы используем \*\*kwargs для передачи переменного количества именованных аргументов. Схоже с \*args, если поставить \*\* перед именем, это имя будет принимать любое количество именованных аргументов. Кортеж/словарь из нескольких переданных аргументов будет доступен под этим именем. Например:

При запуске программы мы увидим следующее:

В этом случае у нас есть функция intro() с параметром \*\*data. В функцию мы передали два словаря разной длины. Затем внутри функции мы прошлись в цикле по словарям, чтобы вывести их содержимое.

Что нужно запомнить:

\*args и \*\*kwargs — специальный синтаксис, позволяющий передавать в функцию переменное количество аргументов. При этом, совсем не обязательно использовать имена аргументов args и kwargs;\*args используется для неименованных аргументов, с которыми можно работать как со списком;\*\*kwargs используется для именованных аргументов, с которыми можно работать как со словарём;если вы хотите использовать и \*args, и \*\*kwargs, то это делается так: func(fargs, \*args, \*\*kwargs), порядок следования аргументов важен.

# **Заключение**

Функции играют ключевую роль в программировании на Python, позволяя создавать структурированный, модульный и повторно используемый код. В этой статье мы рассмотрели основные аспекты определения и объявления функций, что является основой для создания любой программы. Мы также обсудили перегрузку функций и функции с переменным количеством аргументов, которые предоставляют дополнительные возможности для более гибкого и мощного программирования.

Перегрузка функций позволяет использовать одно и то же имя для разных реализаций функции, что улучшает читаемость кода и облегчает его поддержку. Функции с переменным количеством аргументов, такие как \*args и \*\*kwargs, предоставляют возможность передавать неопределенное количество аргументов, что делает функции более универсальными и адаптируемыми к различным ситуациям.

Понимание и умение эффективно использовать эти возможности позволяют создавать более сложные и эффективные приложения, а также обеспечивают лучшее управление кодом. Надеемся, что эта статья помогла вам углубить знания о функциях в Python и вдохновила на использование их возможностей для решения разнообразных задач программирования.

# **Список литературы**

1. https://education.yandex.ru/handbook/python/article/funkcii-oblasti-vidimosti-peredacha-parametrov-v-funkcii

2. https://docs-python.ru/tutorial/opredelenie-funktsij-python/

3. https://docs-python.ru/tutorial/opredelenie-funktsij-python/peregruzka-funktsij/

4. https://habr.com/ru/companies/otus/articles/725374/

5. https://letpy.com/python-guide/functions/

6. https://tproger.ru/translations/python-args-and-kwargs

7. https://sky.pro/wiki/python/mnozhestvennaya-peregruzka-funktsiy-v-python-retsept-dlya-igrovykh-pul/

8. https://pythonworld.ru/tipy-dannyx-v-python/vse-o-funkciyax-i-ix-argumentax.html