Python VIII:
Функции

Функция - это именованный блок кода, который может быть вызван из другого места в программе. Функции могут принимать аргументы (входные данные), обрабатывать их и возвращать результат.

Создание функции начинается с ключевого слова def, за которым следует имя функции, а затем в скобках указываются аргументы функции. Функция может быть определена без аргументов, например:

```
def say_hello():
    print("Hello!")
```

Чтобы вызвать функцию, достаточно написать ее имя и круглые скобки:

```
say_hello()
```

Это вызовет функцию say_hello(), и в результате на экран будет выведено "Hello!".

Функция может иметь любое количество аргументов, их нужно указывать через запятую в скобках. Например:

```
def greet(name):
    print(f"Hello, {name}!")
```

Эта функция принимает один аргумент - имя - и выводит сообщение с приветствием.

```
greet("Alice")
```

Этот код вызовет функцию greet() и передаст ей аргумент "Alice". Результатом будет сообщение "Hello, Alice!".

Функции могут возвращать значение с помощью ключевого слова return. Например:

```
def add(a, b):
    return a + b
```

Эта функция принимает два аргумента a и b и возвращает их сумму.

```
result = add(2, 3)
print(result)
```

Этот код вызовет функцию add() с аргументами 2 и 3, результатом ее работы будет 5, который будет сохранен в переменной result и выведен на экран.

Кроме того, функции могут иметь значение по умолчанию для аргументов. Если функция вызывается без указания какого-либо аргумента, то будет использовано значение по умолчанию. Например:

```
def multiply(a, b=2):
    return a * b
```

Эта функция принимает два аргумента a и b, при этом значение b по умолчанию равно 2.

```
result1 = multiply(3)
result2 = multiply(3, 4)
print(result1, result2)
```

Этот код вызовет функцию multiply() с аргументом 3, а также с аргументами 3 и 4. Результатом первого вызова будет 6, а второго - 12. В результате выполнения этого кода на экран будет выведено "6 12".

1. В Python функции можно определять внутри других функций. Такие функции называются вложенными. Вложенная функция может использовать переменные из внешней функции.
2. В Python функции можно передавать как параметры другим функциям и возвращать из функций как значения.
3. В Python можно определять аргументы функций со значениями по умолчанию. Если аргумент не будет передан при вызове функции, он будет иметь значение, указанное по умолчанию.
4. В Python можно определять функции с переменным числом аргументов. Например, функция может принимать любое количество аргументов, указанных при вызове функции.
5. В Python есть также анонимные функции, которые можно определять с помощью ключевого слова lambda. Анонимные функции не имеют имени и могут содержать только одно выражение. Они обычно используются в качестве аргументов для других функций.
6. В Python функции можно определять внутри классов. Такие функции называются методами. Методы могут использовать переменные и методы класса.

Вложенные функции - это функции, которые определены внутри другой функции. Они имеют доступ к переменным внешней функции и могут быть использованы для создания замыканий.

Замыкание (closure) - это функция, которая запоминает значения из внешнего лексического контекста, в котором она была определена, и может использовать их в своей работе. Это означает, что замыкание имеет доступ к переменным, определенным внутри своей функции-родителя, даже после того, как родительская функция завершена.

Замыкания могут быть полезны в различных ситуациях, например, для создания локальных переменных, которые будут использоваться внутри функции-обработчика, но не должны быть доступны извне.

```
def outer_func(x):
    def inner_func(y):
        return x + y
    return inner_func

closure_func = outer_func(5)
print(closure_func(3))  # Output: 8
```

В этом примере функция inner_func определена внутри функции outer_func и имеет доступ к переменной x, которая определена внутри outer_func. Мы можем вызвать outer_func с аргументом 5 и сохранить возвращаемое значение в переменной closure_func. Затем мы можем вызвать closure_func с аргументом 3, и она вернет 8, так как x равно 5, а y равно 3. Это и есть замыкание: inner_func запоминает значение переменной x из внешней функции outer_func и использует его при каждом вызове.

Лямбда-функции - это способ определения функций, которые могут содержать только одно выражение. Они также известны как анонимные функции, так как они не требуют определения имени функции.

Лямбда-функции могут быть полезны в тех случаях, когда вам нужно передать простую функцию в качестве аргумента другой функции, например при использовании функции map(), filter() и reduce(), или когда вы хотите определить функцию, которая будет использована только один раз.

```
 lambda arguments: expression
```

Где arguments - это список аргументов через запятую, а expression - это выражение, которое вы хотите вычислить в функции.

Например, определим простую лямбда-функцию, которая будет возвращать квадрат переданного числа:

```
square = lambda x: x**2
 print(square(3)) # Output: 9
```

Также, лямбда-функции могут содержать несколько аргументов:

```
multiply = lambda x, y: x*y
 print(multiply(2, 3)) # Output: 6
```

Лямбда-функции также могут использоваться вместо определения простых функций, например, вот так:

```
def apply_func(x, func):
      return func(x)

 print(apply_func(3, lambda x: x**2)) # Output: 9
```

Хотя лямбда-функции могут быть полезны, но их следует использовать с осторожностью, чтобы избежать усложнения кода. Кроме того, лямбда-функции не могут содержать несколько выражений и не поддерживают оператор return.

Декораторы

Декораторы в Python - это функции, которые принимают другую функцию в качестве аргумента, выполняют какие-то действия с этой функцией и возвращают её обратно. С помощью декораторов можно добавлять какие-то общие функции для группы функций, не изменяя код этих функций.

Декораторы определяются в виде функций, которые принимают один аргумент - функцию, которую нужно декорировать. Обычно они возвращают новую функцию, которая заменяет исходную функцию.

Пример декоратора, который просто выводит имя функции:

```
def print_func_name(func):
      def wrapper(*args, **kwargs):
       print("Вызывается функция: ", func.__name__)
       return func(*args, **kwargs)
       return wrapper
```

Здесь определяется функция print_func_name, которая принимает один аргумент func. Внутри определяется новая функция wrapper, которая выводит имя функции func, а затем вызывает саму функцию func. В конце функция wrapper возвращает результат вызова функции func.

Чтобы применить этот декоратор к функции, нужно перед названием функции поставить символ @, а затем название декоратора. Например:

```
@print_func_name
 def my_function():
       print("Привет, я функция!")
```

Пример декоратора, который печатает время выполнения функции:

```
import time

def timer_decorator(func):
    def wrapper(*args, **kwargs):
    start_time = time.time()
    result = func(*args, **kwargs)
    end_time = time.time()
    print(f"Function {func.__name__} took {end_time - start_time}
seconds to run.")
    return result
    return wrapper

@timer_decorator
def my_function():
   time.sleep(2)
   return "Hello, world!"

print(my_function())  # "Hello, world!"
# "Function my_function took 2.000123 seconds to run."
```

В этом примере мы определили декоратор timer_decorator, который принимает функцию func в качестве аргумента и возвращает новую функцию wrapper. Функция wrapper вызывает func, замеряет время выполнения и возвращает результат. Затем мы применили декоратор timer_decorator к функции my_function с помощью символа "@" перед именем декоратора. При вызове функции my_function будет автоматически вызван и декоратор timer_decorator, который измерит время выполнения функции и выведет результат.

Параметризованные декораторы
Декораторы могут быть параметризованными, то есть принимать аргументы. Для этого нужно создать еще один уровень функции-декоратора, который будет принимать аргументы и возвращать функцию-декоратор. Например:

```
 def repeat(n):
      def decorator(func):
      def wrapper(*args, **kwargs):
          for i in range(n):
               func(*args, **kwargs)
     return wrapper
     return decorator
```

В этом примере мы создаем декоратор repeat, который принимает аргумент n и возвращает декоратор decorator. decorator принимает функцию func и возвращает обертку wrapper, которая вызывает func n раз.

Стек декораторов
Декораторы могут быть использованы в стеке, то есть один декоратор может вызывать другой декоратор. Например:

```
def bold(func):
     def wrapper():
     return '<b>' + func() + '</b>'
     return wrapper

def italic(func):
     def wrapper():
     return '<i>' + func() + '</i>'
     return wrapper

@bold
@italic
def hello():
     return 'Hello, world!'
```

В этом примере мы создаем два декоратора, bold и italic, каждый из которых добавляет HTML-теги к возвращаемому значению функции. Мы затем применяем эти декораторы к функции hello в обратном порядке с помощью синтаксиса @bold и @italic. При вызове функции hello.

Рекурсия

Рекурсия - это когда функция вызывает сама себя. То есть, функция может выполнить определенные действия, затем вызвать саму себя и повторить те же действия. Это похоже на то, как матрешка содержит внутри себя другую матрешку.

Рекурсия очень полезна в тех случаях, когда нужно многократно повторять один и тот же процесс или когда задача может быть разбита на более простые подзадачи.

Например, давай представим, что у тебя есть лабиринт и тебе нужно найти выход. Как ты можешь это сделать? Можно идти по коридорам и поворачивать в разные стороны, пока не найдешь выход. Это называется итеративным подходом.

А можно попытаться разбить задачу на более маленькие части. Например, можно идти по коридору, пока не дойдешь до перекрестка, а затем попробовать пойти по всем возможным направлениям и продолжать так до тех пор, пока не найдешь выход. Это называется рекурсивным подходом.

В программировании рекурсия используется для обхода деревьев и графов, сортировки, поиска и т.д.

Например, давай рассмотрим функцию, которая находит факториал числа:

```
def factorial(n):     if n == 0:
      return 1
      else:
      return n * factorial(n-1)
```

Здесь функция factorial() вызывает саму себя с аргументом n-1, пока n не достигнет нуля. Когда n равен нулю, функция возвращает 1, что является базовым случаем рекурсии.

Но стоит заметить, что факториал можно вычислить более эффективными способами, например, используя цикл. Так как на больших числах эта функция будет очень не эффективна.

Одним из важных моментов при использовании рекурсии является обеспечение того, чтобы базовый случай был достигнут в какой-то момент. В противном случае, функция будет вызывать саму себя бесконечное число раз, что приведет к ошибке переполнения стека (stack overflow).

Дополнительно о функциях!

Функции могут быть определены внутри условных выражений и циклов:

```
 def some_function():
      if some_condition:
      def inner_function():
            print("Inner function")
      inner_function()
```

В этом примере, если условие some_condition истинно, то внутри функции some_function будет определена вложенная функция inner_function, которая будет вызвана сразу же после определения. Важно понимать, что inner_function будет видна только внутри блока if и недоступна за его пределами.

Функции также могут быть определены в качестве аргументов других функций. Такие функции называются функциями высшего порядка:

```
def apply_func(func, x):
    return func(x)
```

```
def square(x):
    return x**2

print(apply_func(square, 5)) # Выведет 25
```

В этом примере функция apply_func принимает два аргумента: функцию func и значение x. Функция func применяется к x, и результат возвращается из apply_func. В примере square передается в качестве аргумента func, и она применяется к значению 5. Результатом является 25.

Область видимости (global, nonlocal):

global и nonlocal - это ключевые слова в Python, которые используются для работы с областью видимости переменных внутри функций.

global используется для объявления переменной, которая должна использоваться внутри функции и в глобальной области видимости (вне функции). Таким образом, при использовании global внутри функции, вы можете изменять значения глобальных переменных.
Пример:

```
x = 10

def func():
    global x
    x = 20
    print(x)

func()  # Вывод: 20
print(x)  # Вывод: 20
```

nonlocal используется для работы с переменными, которые находятся в области видимости выше текущей функции. Таким образом, вы можете использовать nonlocal для изменения значения переменной, которая определена в функции, внутри вложенной функции.
Пример:

```
def outer():
    x = 10
    def inner():
        nonlocal x
        x = 20
        print(x)
    inner()
    print(x)
outer()  # Вывод: 20, 20
```

locals() - это встроенная функция Python, которая возвращает словарь, содержащий все локальные переменные в текущей области видимости.

vars() - это встроенная функция Python, которая возвращает словарь, содержащий все атрибуты объекта (если они есть). Если аргумент не указан, vars() возвращает словарь, содержащий локальные переменные в текущей области видимости.

dir() - это встроенная функция Python, которая возвращает список всех атрибутов и методов объекта (если они есть).

id() - это встроенная функция Python, которая возвращает уникальный идентификатор объекта.