InnoDom

Функции

Принципы функционального программирования

Функциональное программирование – это способ создания программ, используя функции – части кода, которые можно использовать повторно в любой части кода.

Принципы функционального программирования:

- функции должны быть независимые, функция всегда работает с принимающими значениями и выдает результат своей работы.
- переменные не изменяемые, не рекомендуется изменять переменную после её инициализации.
- рекомендуется писать прозрачные функции, если вы можете заменить вызов функции на возвращаемое значение, а состояние функции не измениться, то функция прозрачна.

Именованные функции

Именованная функция – это функция, которая заключает в себе код под именем для выполнения **одного действия**.
Функция может принимать аргументы, а может не принимать и такие функции в большинстве случаев являются функциями-генераторами.

Функция должна возвращать **результат** своего выполнения, если она ничего не возвращает, то такие функции называют процедурами. Обычно процедуры используются, когда нужно сделать вывод какой-либо структуры либо запускающие программу основные функции, чаще всего их называют как *main*.

Пример создания процедуры:

```
def msg():
    print("Hello world!")
```

Пример создания функции:

```
def msg():
    return "hello world!"


print(msg())
```

Аргументы функции

Функция может принимать в себя аргументы (параметры), это те элементы с которыми функция будет работать. В Python у функций существует 2 типа аргументов: `*args` и `**kwargs`.

Args - это обычные аргументы, которые мы передаем в функцию.

```
def summ(num1, num2):
    return num1+num2


print(summ(4, 5))
```

! такие аргументы обязательно записываются в **строгом порядке** и передаётся тип данных, который предположительно должен быть указан. К примеру, если передадим в функцию `sum()` число и строку, эта функция вызовет исключение из-за складывания элементов разных типов.

Аргументы функции

Kwargs - это ключевые аргументы которые мы передаем в функцию.

```
def summ(num1, num2):
    return num1+num2


print(summ(num1=4, num2=5))
```

! ключевые аргументы могут указываться в любом порядке.

Аргументы функции

Можно передавать и те, и те аргументы одновременно.

```
def summ(num1, num2):
    return num1+num2


print(summ(5, num2=4))
```

! Когда указываем разные по характеру аргументы, то сначала передаем **args**-аргументы, а только потом **kwargs**.

Аргументы функции

Аргументы могут быть заданы по умолчанию, в случае, если такой параметр не будет передан в функцию, она возьмет значение по умолчанию этого аргумента.

```
def summ(num1, num2=9):
    return num1+num2


print(summ(5))
```

! Всегда, с самого начала, указываются аргументы **без**, а потом **с значениями по умолчанию**, иначе исключение.

Args и kwargs

Args и **kwargs** позволяют забрать незарезервированные аргументы, передаваемые в функцию.
Kwargs позволяет забрать ключевые аргументы и передавать их в виде словаря, **args** - в виде списка.

```
def summ(num1, num2=9, *args, **kwargs):
    print(args, kwargs)
    return num1 + num2


print(summ(5, 1, 2, 3, 5, 4, 3, 7, 4, 2, 2, 1, 5, g=3, d=2, c=24, b=12, a=54))
```

Локальные и глобальные переменные

Все переменные, которые мы объявляем в функции, включая её аргументы, являются **локальными**, а это значит, что обратиться к ним можно только внутри функции, за функцией они не существуют.

Глобальные переменные объявляются внутри функции и работают, а также хранят значения полученные в функции за ней, учтите что использовать такие переменные не всегда безопасно.

```
def summ(num1, num2=9):
    global a
    a = 11
    return num1+num2


print(summ(5), a)
```

return

Когда функция возвращает какие-либо данные, она **завершает** своё выполнение. В зависимости от условия, функция может возвращать разные данные.

```
def myfunc(a):
    if a < 100:
        return "a меньше 100"
    else:
        return "a больше 100"


print(myfunc(4))
```

return

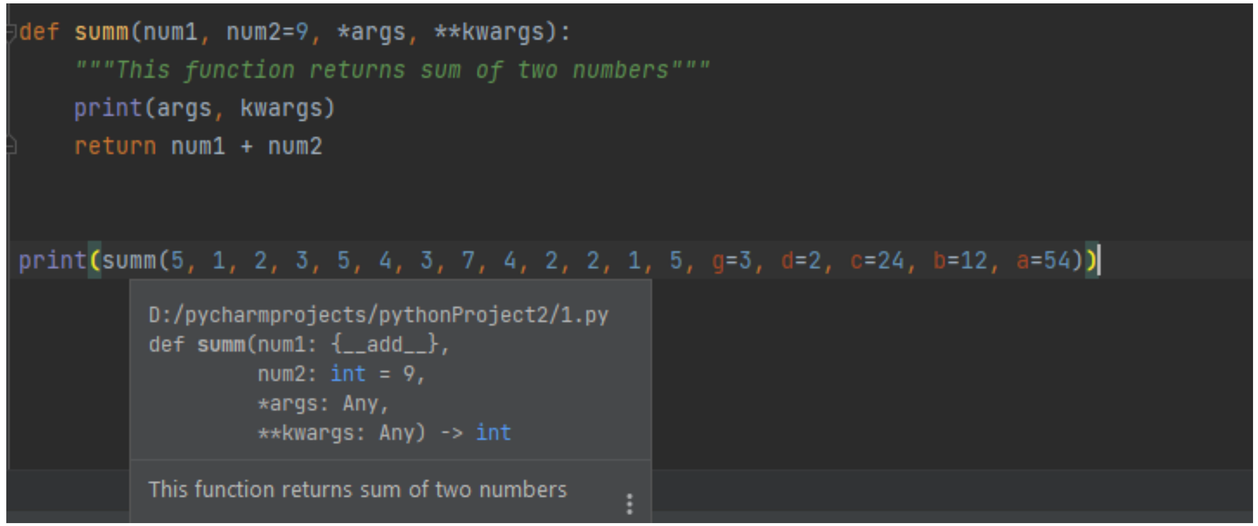
Также `return` может возвращать несколько значений в виде **кортежа**, все значения перечисляются через запятую.

```
def myfunc(a, b):
    return a, b


print(myfunc(4, 5))
```

Документирование функций

Python позволяет для функций создавать строки документации. Они позволяют быстро и просто документировать ваши функции.



Рекурсия

Рекурсия – это подход в программировании, при котором функция вызывает сама себя. Любую рекурсивную функцию можно представить как цикл, а цикл можно представить как рекурсию.

Преимущества рекурсии:

- каждый новый вызов функции в функции начинается с чистого листа, а значения из предыдущих вызовов отдаются в стек.
- код, написанный через рекурсию читается проще. Решения выглядит лучше.

Рекурсия

Недостаток рекурсии:

- размер стека, которому проталкиваются элементы из каждого вызова не безграничен.
- рекурсию трудно представить, если рекурсия спроектирована плохо, она может убить ваше приложение или сильно его замедлить.

```
def listCrawl(list1):
    if len(list1) > 0:
        print(list1[len(list1)-1])
        return listCrawl(list1[0:len(list1) - 1])
    else:
        return None


listCrawl([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

Рекурсия

Правила создания рекурсии:

- рекурсия не должна создавать больше чем 3000 слоев (вызовов)
- у рекурсии всегда должно быть условие остановки
- использовать рекурсию, для того чтобы решение становилось меньше и понятнее
- использовать рекурсию, когда знаешь глубину вызов
- если код читает другой человек, выбор в пользу рекурсии (код красивее и понятнее)
- если скорость не так важна выбор в пользу рекурсии