**Тема занятия №6: Функции. Модули, библиотеки и пакеты**

Начнем с определения, **функция** — это фрагмент программного кода, который решает какую-либо задачу. Мы можем выделить основные преимущества, при работе с функцией, такие как:

* функцию можно вызывать в любом месте основной программы;
* функции помогают избегать дублирования кода при многократном его использовании.

Ранее мы уже встречались с функциями, и работаем с ними. В частности, функция print (), которая выводит некоторое значение на консоль. Python имеет множество встроенных функций и позволяет определять свои функции.

Рассмотрим простой пример: Вы торгуете мёдом, и после каждой продажи вам нужно печатать чек. В нём должно быть указано: название фирмы, дата продажи, список наименований проданных товаров, их количество, цены, общая сумма, а также сакраментальная фраза "Спасибо за покупку!".

Если не пользоваться функциями, всё придётся прописывать вручную. В простейшем случае программа будет выглядеть так:

**print("ООО Медовый Гексагон")**

**print("Мёд липовый", end=" ")**

**print(1, end="шт ")**

**print(1250, end="р")**

**print("\nCумма", 1250, end="р")**

**print("\nСпасибо за покупку!")**

А теперь представьте, что произойдёт, когда вы раскрутитесь, и покупатели станут приходить один за другим. В таком случае, чеки надо будет выдавать очень быстро. Но что делать, если вдруг нагрянет ваш любимый клиент и купит 10 сортов мёда в разных количествах? Далеко не все в очереди согласятся ждать, пока вы посчитаете общую сумму и внесёте её в чек.

Хорошо, что данный процесс можно легко оптимизировать с использованием функций**.**

**def print\_check(honey\_positions):**

**sum = 0 # переменная для накопления общей суммы**

**print("ООО Медовый Гексагон\n")**

**# в цикле будем выводить название, количество и цену**

**for honey in honey\_positions:**

**name = honey[0]**

**amount = honey[1]**

**price = honey[2]**

**print(f"{name} ({amount} шт.) - {price} тг.")**

**sum += amount \* price # здесь же будем считать ещё и общую сумму**

**print(f"\nИтого: {sum} тг.")**

**print("Спасибо за покупку!")**

Встаёт вопрос: где же обещанное упрощение и куда подевались товары? Как раз для этого, мы и будем описывать состав покупки не напрямую в функции, а в отдельном списке кортежей. Каждый кортеж состоит из трёх элементов: название товара, количество и цена.

**# (название, количество, цена за штуку)**

**honey\_positions = [**

**("Мёд липовый", 3, 1250),**

**("Мёд цветочный", 7, 1000),**

**("Мёд гречишный", 6, 1300),**

**("Донниковый мёд", 1, 1750),**

**("Малиновый мёд", 10, 2000),**

**]**

Теперь этот список передадим в функцию как аргумент, и самостоятельно считать больше не придётся.

**print\_check(honey\_positions)**

Результат программы ниже:

**>**

**ООО Медовый Гексагон**

**Мёд липовый (3 шт.) - 1250 тг.**

**Мёд цветочный (7 шт.) - 1000 тг.**

**Мёд гречишный (6 шт.) - 1300 тг.**

**Донниковый мёд (1 шт.) - 1750 тг.**

**Малиновый мёд (10 шт.) - 2000 тг.**

**Итого: 40300 тг.**

**Спасибо за покупку!**

Определение функции начинается с выражения def, которое состоит из имени функции, набора скобок с параметрами и двоеточия. Параметры в скобках необязательны. А со следующей строки идет блок инструкций, которые выполняет функция. Все инструкции функции имеют отступы от начала строки.

Например, определение простейшей функции:

**def say\_hello():**

**print("Hello")**

Функция называется say\_hello. Она не имеет параметров и содержит одну единственную инструкцию, которая выводит на консоль строку "Hello".

Обратите внимание, что инструкции функции должны иметь отступы от начала функции. Например:

**def say\_hello():**

**print("Hello")**

**print("Bye")**

Здесь инструкция print("Bye") не имеет отступов от начала функции say\_hello и поэтому в эту функцию не входит. Обычно между определением функции и остальными инструкциями, которые не входят в функцию, располагаются две пустых строки.

Для вызова функции указывается имя функции, после которого в скобках идет передача значений для всех ее параметров:

имя\_функции ([параметры])

Например, определим и вызовем функцию:

**def say\_hello(): # определение функции say\_hello**

**print("Hello")**

**say\_hello() # вызов функции say\_hello**

**say\_hello()**

**say\_hello()**

Здесь три раза подряд вызывается функция say\_hello. В итоге мы получим следующий консольный вывод:

**Hello**

**Hello**

**Hello**

Обратите внимание, что функция сначала определяется, а потом вызывается.

Если функция имеет одну инструкцию, то ее можно разместить на одной строке с остальным определением функции:

**def say\_hello():**

**print("Hello")**

**say\_hello()**

Подобным образом можно определять и вызывать и другие функции. Например, определим и выполним несколько функций:

**def say\_hello():**

**print("Hello")**

**def say\_goodbye():**

**print("Good Bye")**

**say\_hello()**

**say\_goodbye()**

Консольный вывод:

**Hello**

**Good Bye**

**Локальные функции**

Одни функции могут определяться внутри других функций - внутренние функции еще называют локальными. Локальные функции можно использовать только внутри той функции, в которой они определены. Например:

**def print\_messages():**

**# определение локальных функций**

**def say\_hello(): print("Hello")**

**def say\_goodbye(): print("Good Bye")**

**# вызов локальных функций**

**say\_hello()**

**say\_goodbye()**

**# Вызов функции print\_messages**

**print\_messages()**

**#say\_hello() # вне функции print\_messages функция say\_hello не доступна**

Здесь функции say\_hello() и say\_goodbye() определены внутри функции print\_messages() и поэтому по отношению к ней являются локальными. Соответственно они могут использоваться только внутри функции print\_messages().

Функция может принимать параметры. Через параметры в функцию можно передавать данные. Банальный пример - функция print(), которая с помощью параметра принимает значение, выводимое на консоль.

Теперь определим и используем свою функцию с параметрами:

**def say\_hello(name):**

**print(f"Hello, {name}")**

**say\_hello("Tom")**

**say\_hello("Bob")**

**say\_hello("Alice")**

Функция say\_hello имеет параметр name, и при вызове функции мы можем передать этому параметру какой-либо значение. Внутри функции мы можем использовать параметр как обычную переменную, например, вывести значение этого параметра на консоль функцией print. Так, в выражении:

**say\_hello("Tom")**

Строка "Tom" будет передаваться параметру name. В итоге при выполнении программы мы получим следующий консольный вывод:

**Hello, Tom**

**Hello, Bob**

**Hello, Alice**

При вызове функции значения передаются параметрам по позиции. Например, определим и вызовем функцию с несколькими параметрами:

**def print\_person(name, age):**

**print(f"Name: {name}")**

**print(f"Age: {age}")**

**print\_person("Tom", 37)**

Здесь функция print\_person принимает два параметра: name и age. При вызове функции:

**print\_person("Tom", 37)**

Первое значение - "Tom" передается первому параметру, то есть параметру name. Второе значение - 37 передается второму параметру - age. И внутри функции значения параметров выводятся на консоль:

**Name: Tom**

**Age: 37**

**Аргументы**

**Позиционные**

Вспомним, аргумент — это конкретное значение, которое передаётся в функцию. Аргументом может быть любой объект. Он может передаваться, как в литеральной форме, так и в виде переменной.

Значения в позиционных аргументах подставляются согласно позиции имён аргументов:

**nums = [42, 11, 121, 13, 7]**

**state = True**

**# в данном примере**

**# 1-я позиция "nums" -> parameter\_1**

**# 2-я позиция "state" -> parameter\_2**

**def test\_params(parameter\_1, parameter\_2):**

**pass**

**# равнозначные варианты вызова функции**

**test\_params(nums, state)**

**test\_params([42, 11, 121, 13, 7], True)**

**Именованные**

Пусть есть функция, принимающая три аргумента, а затем выводящая их на экран. Python позволяет явно задавать соответствия между значениями и именами аргументов.

**def print\_trio(a, b, c):**

**print(a, b, c)**

**print\_trio(c=4, b=5, a=6)**

**> 6 5 4**

При вызове соответствие будет определяться по именам, а не по позициям аргументов.

Необязательные параметры (параметры по умолчанию)

Python позволяет делать отдельные параметры функции необязательными. Если при вызове значение такого аргумента не передается, то ему будет присвоено значение по умолчанию.

**def not\_necessary\_arg(x='My', y='love'):**

**print(x, y)**

**# если не передавать в функцию никаких значений, она отработает со значениями по умолчанию**

**not\_necessary\_arg()**

**> My love**

**# переданные значения заменяют собой значения по умолчанию**

**not\_necessary\_arg(2, 1)**

**> 2 1**

**Аргументы переменной длины (args, kwargs)**

Когда заранее неизвестно, сколько конкретно аргументов будет передано в функцию, мы пользуемся аргументами переменной длины. Звёздочка "\*" перед именем параметра сообщает интерпретатору о том, что количество позиционных аргументов будет переменным:

**def infinity(\*args):**

**print(args)**

**infinity(42, 12, 'test', [6, 5])**

**> (42, 12, 'test', [6, 5])**

Переменная args составляет кортеж из переданных в функцию аргументов.

Функции в питоне могут также принимать и переменное количество именованных аргументов. В этом случае перед названием параметра ставится "\*\*":

**def named\_infinity(\*\*kwargs):**

**print(kwargs)**

**named\_infinity(first='nothing', second='else', third='matters')**

**> {'first': 'nothing', 'second': 'else', 'third': 'matters'}**

Здесь kwargs уже заключает аргументы не в кортеж, а в словарь.

Функции в Python способны возвращать любой тип объекта.

В Питоне поддерживается возврат функциями сразу несколько значений. Достаточно перечислить их через запятую после инструкции return. Возвращаемым типом будет кортеж (tuple), который можно распаковать в переменные.

**def calculate(num1, num2):**

**return num1 + num2, num1 - num2, num1 \* num2**

**# для так называемой распаковки нескольких значений**

**# их следует присвоить равному количеству аргументов**

**res1, res2, res3 = calculate(7, 6)**

**print(res1, res2, res3)**

**> 13 1 42**

**print(type(calculate(7, 6)))**

**<class 'tuple'>**

Обратите внимание, что количество возвращаемых значение в кортеже должно совпадать с количеством переменных при распаковке. Иначе произойдет ошибка:

**def calculate(num1, num2):**

**return num1 + num2, num1 - num2**

**# для так называемой распаковки нескольких значений**

**# их следует присвоить равному количеству аргументов**

**res1, res2, res3 = calculate(7, 6)**

**print(res1, res2, res3)**

**>**

**ValueError: not enough values to unpack (expected 3, got 2)**

**Пустая функция**

Иногда разработчики оставляют реализацию на потом, и чтобы объявленная функция не генерировала ошибки из-за отсутствия тела, в качестве заглушки используется ключевое слово pass:

**def empty():**

**pass**

Оператор return не только возвращает значение, но и производит выход из функции. Поэтому он должен определяться после остальных инструкций. Например:

**def get\_message():**

**return "Hello METANIT.COM"**

**print("End of the function")**

**print(get\_message())**

С точки зрения синтаксиса данная функция корректна, однако ее инструкция print("End of the function") не имеет смысла - она никогда не выполнится, так как до ее выполнения оператор return возвратит значение и произведет выход из функции.

Однако мы можем использовать оператор return и в таких функциях, которые не возвращают никакого значения. В этом случае после оператора return не ставится никакого возвращаемого значения. Типичная ситуация - в зависимости от опеределенных условий произвести выход из функции:

**def print\_person(name, age):**

**if age > 120 or age < 1:**

**print("Invalid age")**

**return**

**print(f"Name: {name} Age: {age}")**

**print\_person("Tom", 22)**

**print\_person("Bob", -102)**

Здесь функция print\_person в качестве параметров принимает имя и возраст пользователя. Однако в функции вначале мы проверяем, соответствует ли возраст некоторому диапазону (меньше 120 и больше 0). Если возраст находится вне этого диапазона, то выводим сообщение о недопустимом возрасте и с помощью оператора return выходим из функции. После этого функция заканчивает свою работу.

Однако если возраст корректен, то выводим информацию о пользователе на консоль. Консольный вывод:

**Name: Tom Age: 22**

**Invalid age**

Область видимости или scope определяет контекст переменной, в рамках которого ее можно использовать. В Python есть два типа контекста: глобальный и локальный.

Глобальный контекст подразумевает, что переменная является глобальной, она определена вне любой из функций и доступна любой функции в программе. Например:

**name = "Tom"**

**def say\_hi():**

**print("Hello", name)**

**def say\_bye():**

**print("Good bye", name)**

**say\_hi()**

**say\_bye()**

Здесь переменная name является глобальной и имеет глобальную область видимости. И обе определенные здесь функции могут свободно ее использовать.

**Локальный контекст**

В отличие от глобальных переменных локальная переменная определяется внутри функции и доступна только из этой функции, то есть имеет локальную область видимости:

**def say\_hi():**

**name = "Sam"**

**surname = "Johnson"**

**print("Hello", name, surname)**

**def say\_bye():**

**name = "Tom"**

**print("Good bye", name)**

**say\_hi()**

**say\_bye()**

В данном случае в каждой из двух функций определяется локальная переменная name. И хотя эти переменные называются одинаково, но тем не менее это две разных переменных, каждая из которых доступна только в рамках своей функции. Также в функции say\_hi() определена переменная surname, которая также является локальной, поэтому в функции say\_bye() мы ее использовать не сможем.

**Скрытие переменных**

Есть еще один вариант определения переменной, когда локальная переменная скрывают глобальную с тем же именем:

**name = "Tom"**

**def say\_hi():**

**name = "Bob" # скрываем значение глобальной переменной**

**print("Hello", name)**

**def say\_bye():**

**print("Good bye", name)**

**say\_hi() # Hello Bob**

**say\_bye() # Good bye Tom**

Здесь определена глобальная переменная name. Однако в функции say\_hi определена локальная переменная с тем же именем name. И если функция say\_bye использует глобальную переменную, то функция say\_hi использует локальную переменную, которая скрывает глобальную.

Если же мы хотим изменить в локальной функции глобальную переменную, а не определить локальную, то необходимо использовать ключевое слово global:

**name = "Tom"**

**def say\_hi():**

**global name**

**name = "Bob" # изменяем значение глобальной переменной**

**print("Hello", name)**

**def say\_bye():**

**print("Good bye", name)**

**say\_hi() # Hello Bob**

**say\_bye() # Good bye Bob**

**Функции. Рекурсия**

**Рекурсивная функция** — это та, которая вызывает сама себя.

В качестве простейшего примера рассмотрите следующий код:

**def factorial\_recursive(n):**

**if n == 1:**

**return n**

**else:**

**return n\*factorial\_recursive(n-1)**

Вызывая рекурсивную функцию здесь и передавая ей целое число, вы получаете факториал этого числа (n!).

Факториал числа — это число, умноженное на каждое предыдущее число вплоть до 1.

Например, факториал числа 7:

7! = 7\*6\*5\*4\*3\*2\*1 = 5040

Вывести факториал числа можно с помощью функции:

**num = 3**

**print(f"Факториал {num} это {factorial\_recursive(num)}")**

Эта функция выведет: «Факториал 3 это 6». Еще раз рассмотрим эту рекурсивную функцию:

**def factorial\_recursive(n):**

**...**

По аналогии с обычной функцией имя рекурсивной указывается после def, а в скобках обозначается параметр n:

**def factorial\_recursive(n):**

**if n == 1:**

**return n**

**else:**

**return n\*factorial\_recursive(n-1)**

Благодаря условной конструкции переменная n вернется только в том случае, если ее значение будет равно 1. Это еще называют условием завершения. Рекурсия останавливается в момент удовлетворения условиям.

**def factorial\_recursive(n):**

**if n == 1:**

**return n**

**else:**

**return n\*factorial\_recursive(n-1)**

В коде выше выделен фрагмент самой рекурсии. В блоке else условной конструкции возвращается произведение n и значения этой же функции с параметром n-1.

Это и есть рекурсия. В нашем примере это так сработало:

3 \* (3-1) \* ((3-1)-1) # так как 3-1-1 равно 1, рекурсия остановилась

**Детали работы рекурсивной функции**

Чтобы еще лучше понять, как это работает, разобьем на этапы процесс выполнения функции с параметром 3.

Для этого ниже представим каждый экземпляр с реальными числами. Это поможет «отследить», что происходит при вызове одной функции со значением 3 в качестве аргумента:

**# Первый вызов**

**factorial\_recursive(3):**

**if 3 == 1:**

**return 3**

**else:**

**return 3\*factorial\_recursive(3-1)**

**# Второй вызов**

**factorial\_recursive(2):**

**if 2 == 1:**

**return 2**

**else:**

**return 2\*factorial\_recursive(2-1)**

**# Третий вызов**

**factorial\_recursive(1):**

**if 1 == 1:**

**return 1**

**else:**

**return 1\*factorial\_recursive(1-1)**

Рекурсивная функция не знает ответа для выражения 3\*factorial\_recursive(3–1), поэтому она добавляет в стек еще один вызов.

**Как работает рекурсия**

/\ factorial\_recursive(1) - последний вызов

|| factorial\_recursive(2) - второй вызов

|| factorial\_recursive(3) - первый вызов

Выше показывается, как генерируется стек. Это происходит благодаря процессу LIFO (last in, first out, «последним пришел — первым ушел»). Как вы помните, первые вызовы функции не знают ответа, поэтому они добавляются в стек.

Но как только в стек добавляется вызов factorial\_recursive(1), для которого ответ имеется, стек начинает «разворачиваться» в обратном порядке, выполняя все вычисления с реальными значениями. В процессе каждый из слоев выпадает в процессе.

factorial\_recursive(1) завершается, отправляет 1 в

factorial\_recursive(2) и выпадает из стека.

factorial\_recursive(2) завершается, отправляет 2\*1 в

factorial\_recursive(3) и выпадает из стека. Наконец, инструкция else здесь завершается, возвращается 3 \* 2 = 6, и из стека выпадает последний слой.

Рекурсия в Python имеет ограничение в 3000 слоев.

**>>> import sys**

**>>> sys.getrecursionlimit()**

**3000**

Тем не менее рекурсия может быть медленной, если ее неправильно реализовать. Из-за этого вычисления будут происходить чаще, чем требуется.

**def factorial\_iterative(num):**

**factorial = 1**

**if num < 0:**

**print("Факториал не вычисляется для отрицательных чисел")**

**else:**

**for i in range (1, num + 1):**

**factorial = factorial\*i**

**print(f"Факториал {num} это {factorial}")**