**Тема занятия №07: Функции. Модули, библиотеки и пакеты**

Python lambdas - это небольшие, анонимные, суб-синтаксические работы, которые более ограничены, но короче, чем обычные функции Python.

Лямбда-функции - это анонимные функции в Python. Лямбда-функции похожи на обычные функции. Разница между обычной функцией и лямбда-функцией заключается в том, что они могут быть определены без имени, но обычные функции определяются с помощью ключевого слова def.

Ключевое слово lambda используется для определения анонимной или лямбда-функции.

Если мы сравним лямбда-функцию и обычную функцию, то,

Лямбда - функция может принимать любое число аргументов, но может иметь только одно выражение, в то время как обычная функция имеет точное количество аргументов, которые мы объявляем в момент определения.

Лямбда-функция - это однострочные функция. Ее тело содержит выражение в той же строке, в которой оно определено. В обычных функциях, функции содержат блоки тела, в которых определены для выполнения некоторые операторы.

Поскольку лямбда - это однострочная функция, ее можно вызвать мгновенно, но обычная функция должна вызывать сама себя и требовать времени для вызова.

Формальное определение лямбда-выражения:

**lambda [параметры] : инструкция**

Определим простейшее лямбда-выражение:

**message = lambda: print("hello")**

**message() # hello**

Здесь лямбда-выражение присваивается переменной message. Это лямбда-выражение не имеет параметров, ничего не возвращает и просто выводит строку "hello" на консоль. И через переменную message мы можем вызвать это лямбда-выражение как обычную функцию. Фактически оно аналогично следующей функции:

**def message():**

**print("hello")**

Если лямбда-выражение имеет параметры, то они определяются после ключевого слова lambda. Если лямбда-выражение возвращает какой-то результат, то он указывается после двоеточия. Например, определим лямбда-выражение, которое возвращает квадрат числа:

**square = lambda n: n \* n**

**print(square(4)) # 16**

**print(square(5)) # 25**

В данном случае лямбда-выражение принимает один параметр - n. Справа от двоеточия идет возвращаемое значение - n\* n. Это лямбда-выражение аналогично следующей функции:

**def square2(n):**

**return n \* n**

Аналогичным образом можно создавать лямбда-выражения, которые принимают несколько параметров:

**sum = lambda a, b: a + b**

**print(sum(4, 5)) # 9**

**print(sum(5, 6)) # 11**

Хотя лямбда-выражения позволяют немного сократить определения функций, тем не менее они ограничены тем, что они могут выполнять только одну инструкцию. Однако они могут быть довольно удобны в тех случаях, когда необходимо использовать функцию для передачи в качестве параметра или возвращения в другой функции. Например, передача лямбда-выражения в качестве параметра:

**def do\_operation(a, b, operation):**

**result = operation(a, b)**

**print(f"result = {result}")**

**do\_operation(5, 4, lambda a, b: a + b) # result = 9**

**do\_operation(5, 4, lambda a, b: a \* b) # result = 20**

Создается объект Python типа int, который сохраняет значение 5. x же является символом, который ссылается на объект.

Теперь проверим тип x и адрес, на которой он ссылается. Это можно сделать с помощью встроенных функций type и id.

**>>> type(x)**

**<class 'int'>**

**>>> id(x)**

**4308964832**

В итоге x ссылается на объект типа int, а расположен он по адресу, который вернула функция id.

Просто и понятно.

А что происходит при определении вот такой функции:

**>>> def f(x):**

**... return x \* x**

**...**

Лямбда с несколькими аргументами

Определить лямбда-функцию с одним аргументом не составляет труда.

**>>> f = lambda x: x \* x**

**>>> f(5)**

**25**

А если их должно быть несколько, то достаточно лишь разделить значения запятыми. Предположим, что нужна функция, которая берет два числовых аргумента и возвращает их произведение.

**>>> f = lambda x, y: x \* y**

**>>> f(5, 2)**

**10**

Отлично! А как насчет лямбда-функции без аргументов?

Лямбда-функция без аргументов

Допустим, нужно создать функцию без аргументов, которая бы возвращала True. Этого можно добиться с помощью следующего кода.

**>>> f = lambda: True**

**>>> f()**

**True**

Несколько лямбда-функций

В определенный момент возникнет вопрос: а можно ли иметь лямбда-функцию из нескольких строк.

Ответ однозначен: нет.

Лямбда-функции в Python всегда принимают только одно выражение. Если же их несколько, то лучше создать обычную функцию.

Примеры лямбда-функций

Теперь рассмотрим самые распространенные примеры использования лямбда-функций.

Лямбда-функция и map

Распространенная операция со списками в Python — применение операции к каждому элементу.

map() — это встроенная функция Python, принимающая в качестве аргумента функцию и последовательность. Она работает так, что применяет переданную функцию к каждому элементу.

Предположим, есть список целых чисел, которые нужно возвести в квадрат с помощью map.

**>>> L = [1, 2, 3, 4]**

**>>> list(map(lambda x: x\*\*2, L))**

**[1, 4, 9, 16]**

Обратите внимание на то, что в Python3 функция map возвращает объект Map, а в Python2 — список.

Так, вместо определения функции и передачи ее в map в качестве аргумента, можно просто использовать лямбда для быстрого определения ее прямо внутри. В этом есть смысл, если упомянутая функция больше не будет использоваться в коде.

Вот еще один пример.

Лямбда-функция и filter

filter() — это еще одна встроенная функция, которая фильтрует последовательность итерируемого объекта.

Другими словами, функция filter отфильтровывает некоторые элементы итерируемого объекта (например, списка) на основе какого-то критерия. Критерий определяется за счет передачи функции в качестве аргумента. Она же применяется к каждому элементу объекта.

Если возвращаемое значение — True, элемент остается. В противном случае — отклоняется. Определим, например, простую функцию, которая возвращает True для четных чисел и False — для нечетных:

**def even\_fn(x):**

**if x % 2 == 0:**

**return True**

**return False**

**print(list(filter(even\_fn, [1, 3, 2, 5, 20, 21])))**

**#вывод: [2, 20]**

С лямбда-функциями это все можно сделать максимально сжато. Код выше можно преобразовать в такой, написанный в одну строку.

**print(list(filter(lambda x: x % 2 == 0, [1, 3, 2, 5, 20, 21])))**

И в этом сила лямбда-функций.

Пример 1

Можно написать лямбду, которая удваивает свой аргумент: lambda x: x\*2, и использовать её в функции map, чтобы удвоить все элементы в списке:

**my\_list = [1, 2, 3, 4, 5, 6]**

**new\_list = list(map(lambda x: x\*2, my\_list))**

**print(new\_list) # [2, 4, 6, 8, 10, 12]**

Заметьте разницу между этим кодом и функцией double, которую мы написали выше: лямбда гораздо компактнее.

Пример 2

А ещё можно создать лямбда-функцию, которая ищет числа больше нуля: lambda x: x > 0 и использовать в filter, чтобы создать список исключительно положительных чисел.

**my\_list = [18, -3, 5, 0, -1, 12]**

**new\_list = list(filter(lambda x: x > 0, my\_list))**

**print(new\_list) # [18, 5, 12]**

Лямбда-функция определяется там, где используется. Таким образом, в памяти не придётся хранить функцию с именем. Поэтому для одноразового использования целесообразно писать лямбда-функции, чтобы избежать загромождения.

Пример 3

Лямбда также может быть возвращаемым значением другой функции.

Если вам потребуется создать несколько функций, умножающих числа (например, удваивающих и утраивающих), лямбда может помочь.

Вместо объявления множества таких функций можно создать всего лишь одну: multiplyBy. А затем можно вызывать эту функцию по нескольку раз с разными аргументами, чтобы создавать функции умножения на два, на три и т.д.

**def muliplyBy (n):**

**return lambda x: x\*n**

**double = multiplyBy(2)**

**triple = muliplyBy(3)**

**times10 = multiplyBy(10)**

Лямбда-функции – это компактный вариант записи функций, содержащих только одно выражение.

**Модули и пакеты**

**Введение**

Библиотека — (от англ. library) в программировании — сборник подпрограмм или объектов, используемых для разработки программного обеспечения (ПО).

В Python библиотеки называются модулями. Модуль в языке Python представляет отдельный файл с кодом, который можно повторно использовать в других программах.

**PyPI**

PyPI — это центральный репозиторий (хранилище) модулей для языка программирования Python. Он как Play Маркет для Android, App Store для iPhone или CPAN для Perl. Модули в Python Модули в Python устроены по иерархическому принципу — как каталоги в файловой системе. Один модуль может быть вложен в другой, причем вложенность не ограничена (хотя на практике редко бывает больше 4).

Чтобы пользоваться функциями, объектами и классами из модуля, весь этот модуль или его часть нужно подключить к программе — импортировать.

**Импорт модуля**

За импорт в Python отвечает директива import.

from math import pi # Возьмём цисло Пи из библиотеки math

Теперь вам доступна переменная pi.

Модуль, переменную, класс или функцию можно при импорте назвать своим именем — для этого служит ключевое слово as:

from math import pi as число\_пи

число\_пи # => 3.141592653589793

Поскольку в программе на языке Python в именах допустимы буквенные символы любых алфавитов, можно использовать даже греческие буквы:

from math import pi as π

Значения после директивы import можно писать через запятую: from math import sin, cos, tan

Значок «\*» означает, что из библиотеки нужно импортировать всѐ, что доступно:

from math import \*

Впрочем, так делать не рекомендуется, поскольку при таком подходе засоряется пространство имѐн

**dir и help**

Пользуемся двумя полезными функциями:

dir (возвращает список со всем содержимым объекта, модуля и т.д.) и help (показывает справку об использовании данного объекта).

import math

dir(math)

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh','asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh','degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor','fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp','lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin','sinh', 'sqrt', 'tan', 'tanh', 'trunc']

import math

help(math.sin)

Help on built-in function sin in module math: sin(...) sin(x)

Return the sine of x (measured in radians)

При импорте модуля this вы познакомитесь с дзеном Python.

import this

А импорт модуля с антигравитацией откроет в браузере комикс о том, что в Python действительно есть модули на все случаи жизни.

import antigravity

**Ранее рассмотренные модули**

Random module

String module

Operator module

Math module

Datetime module

Functools module

import math

dir(math)

import random dir(random)

date import datetime as dt # тип данных 'дата' (год + месяч + день) my\_date = dt.date(2020, 11, 7)

print(my\_date)

print(dt.date.today())

print(dt.date.today().weekday())

shuffle(): перемешивает список

choice(): возвращает случайный элемент списка

import random

a=[4,5,"9","10","aweawef","vbnvgn"]

random.shuffle(a)

print(a)

print(random.choice(a))

Математические операции с датами и интервалами

import datetime as dt # поддерживаются математические и логические операции для дат

date1 = dt.date(2022, 2, 19)

date2 = dt.date(2020, 10, 7)

delta\_time1 = dt.timedelta(days=4, hours=4)

print(date1 + delta\_time1)

print(date1 - delta\_time1)

print(date1 - date2)

print(date2 > date1)

**Описание**

* **pow(num, power)**: возведение числа num в степень power
* **sqrt(num)**: квадратный корень числа num
* **ceil(num)**: округление числа до ближайшего наибольшего целого
* **floor(num)**: округление числа до ближайшего наименьшего целого
* **factorial(num)**: факториал числа
* **degrees(rad)**: перевод из радиан в градусы
* **radians(grad)**: перевод из градусов в радианы
* **cos(rad)**: косинус угла в радианах
* **sin(rad)**: синус угла в радианах
* **tan(rad)**: тангенс угла в радианах
* **acos(rad)**: арккосинус угла в радианах
* **asin(rad)**: арксинус угла в радианах
* **atan(rad)**: арктангенс угла в радианах
* **log(n, base)**: логарифм числа n по основанию base
* **log10(n)**: десятичный логарифм числа n

**Модуль locale**

При форматировании чисел Python по умолчанию использует англосаксонскую систему, при которой разряды целого числа отделяются друг от друга запятыми, а дробная часть от целой отделяется точкой.

# англосаксонская система

1,234.567

# европейская система

1.234,567

**Синтаксис**

* setlocale(category, locale)

Первый параметр указывает на категорию, к которой применяется функция - к числам, валютам или и числам, и валютам.

LC\_ALL: применяет локализацию ко всем категориям - к форматированию чисел, валют, дат.

LC\_NUMERIC: применяет локализацию к числам

LC\_MONETARY: применяет локализацию к валютам

LC\_TIME: применяет локализацию к датам и времени

LC\_CTYPE: применяет локализацию при переводе символов в верхний или нижний регистр

LC\_COLLIATE: применяет локаль при сравнении строк

* Второй параметр функции setlocale указывает на локальную культуру, которую надо использовать. На ОС Windows можно использовать код станы по ISO из двух символов, например, для США - "us", для Германии - "de", для России - "ru".

**Примеры**

import locale

locale.setlocale(locale.LC\_ALL, "de") # для Windows

number = 12345.6789

formatted = locale.format\_string("%f", number)

print(formatted) # 12345,678900

formatted = locale.format\_string("%.2f", number)

print(formatted) # 12345,68

formatted = locale.format\_string("%d", number)

print(formatted) # 12345

formatted = locale.format\_string("%e", number)

print(formatted) # 1,234568e+04

**Модуль decimal**

number = 0.1 + 0.1 + 0.1

print(number) # 0.30000000000000004

from decimal import Decimal

number = Decimal("0.1")

number = number + number + number

print(number) # 0.3

**Округление чисел**

from decimal import Decimal

number = Decimal("0.444")

number = number.quantize(Decimal("1.00"))

print(number) # 0.44

number = Decimal("0.555678")

print(number.quantize(Decimal("1.00"))) # 0.56

number = Decimal("0.999")

print(number.quantize(Decimal("1.00"))) # 1.00

**NumPy**

* NumPy — это расширение языка Python, добавляющее поддержку больших многомерных массивов и матриц, вместе с большой библиотекой высокоуровневых математических функций для операций с этими массивами. (научные вычисления)



>>> import numpy as np

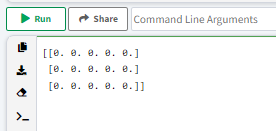
>>> a = np.array([1, 2, 3])

>>> a

array([1, 2, 3])

>>> type(a)

<class 'numpy.ndarray'>

import numpy as np

a=np.zeros((3, 5))

print(a)

np.ones((2, 2, 2))

np.eye(5)

np.empty((3, 3))

numpy.zeroes()

numpy.zeros((rows, columns), dtype)

Эта функция создаст массив numpy с заданным количеством измерений, где каждый элемент будет равняться 0. Если dtype не указан, по умолчанию будет использоваться dtype.

>>> np.arange(10, 30, 5)

array([10, 15, 20, 25])

>>> np.arange(0, 1, 0.1)

array([ 0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9])

>>> np.linspace(0, 2, 9) # 9 чисел от 0 до 2 включительно

array([ 0. , 0.25, 0.5 , 0.75, 1. , 1.25, 1.5 , 1.75, 2. ])

Эта функция создаст массив numpy, элементы которого лежат в диапазоне значений между start до stop, а num\_of\_elements — это размер массива. Тип по умолчанию — float64.

print(np.arange(0, 3000, 1))

[ 0 1 2 ..., 2997 2998 2999]

np.set\_printoptions(threshold=np.nan) – для всего массива

ndarray.ndim

Возвращает количество измерений массива.

import numpy as np

a = np.array([[1,2,3],[4,5,6]])

print(a.ndim)

Вывод кода сверху будет 2, поскольку «a» — это 2-мерный массив.

ndarray.shape

Возвращает кортеж размера массива, то есть (n,m), где n — это количество строк, а m — количество колонок.

import numpy as np

a = np.array([[1,2,3],[4,5,6]])

print(a.shape)

Вывод кода — (2,3), то есть 2 строки и 3 колонки.

ndarray.size

Возвращает общее количество элементов в массиве.

import numpy as np

a = np.array([[1,2,3],[4,5,6]])

print(a.size)

Вывод — 6, потому что 2 х 3.

ndarray.dtype

Возвращает объект, описывающий тип элементов в массиве.

import numpy as np

a = np.array([[1,2,3],[4,5,6]])

print(a.dtype)

Вывод — «int32», поскольку это 32-битное целое число.

Можно явно определить тип данных массива NumPy.

import numpy as np

a = np.array([[1,2,3],[4,5,6]], dtype = float)

print(a.dtype)

Этот код вернет float64, потому что это 64-битное число с плавающей точкой.

**numpy diag()**

import numpy as np

a = np.matrix([[1, 2, 3], [4, 5, 6], [9, 8, 7]])

print("Main diagonal : ",np.diag(a))

**Печать верхней диагонали главной диагонали**

import numpy as np

a = np.matrix([[1, 2, 3], [4, 5, 6], [9, 8, 7]])

print(a)

print("Main diagonal : ",np.diag(a,1))

**Печать нижней диагонали главной диагонали**

import numpy as np

a = np.matrix([[1, 2, 3], [4, 5, 6], [9, 8, 7]])

print(a)

print("Main diagonal : ",np.diag(a,-1))

**Построение диагонали из массива**

#importing numpy library

import numpy as np

a = np.array([5, 6, 7, 8])

print(a)

print("Diagonal : ",np.diag(a))

**Модуль Collections**

Модуль collections - предоставляет специализированные типы данных, на основе словарей, кортежей, множеств, списков.

**collections.Counter** - вид словаря, который позволяет нам считать количество неизменяемых объектов.

**from collections import Counter**

import collections

c = collections.Counter()

for word in ['spam', 'egg', 'spam', 'counter', 'counter', 'counter']:

c[word] += 1print(c)

**elements()**

возвращает список элементов в лексикографическом порядке.

c = Counter(a=4, b=2, c=0, d=-2)

print(list(c.elements()))

['a', 'a', 'a', 'a', 'b', 'b']

**most\_common**([n]) - возвращает n наиболее часто встречающихся элементов, в порядке убывания встречаемости. Если n не указано, возвращаются все элементы.

>>> Counter('abracadabra').most\_common(3)

[('a', 5), ('r', 2), ('b', 2)]

**subtract**([iterable-or-mapping]) - вычитание

>>> c = Counter(a=4, b=2, c=0, d=-2)

>>> d = Counter(a=1, b=2, c=3, d=4)

>>> c.subtract(d)

Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})

**Наиболее часто употребляемые шаблоны для работы с Counter:**

* sum(c.values()) - общее количество.
* c.clear() - очистить счётчик.
* list(c) - список уникальных элементов.
* set(c) - преобразовать в множество.
* dict(c) - преобразовать в словарь.
* c.most\_common()[:-n:-1] - n наименее часто встречающихся элементов.
* c += Counter() - удалить элементы, встречающиеся менее одного раза.

Counter также поддерживает сложение, вычитание, пересечение и объединение:

>>> c = Counter(a=3, b=1)

>>> d = Counter(a=1, b=2)

>>> c + d

Counter({'a': 4, 'b': 3})

>>> c - d

Counter({'a': 2})

>>> c & d

Counter({'a': 1, 'b': 1})

>>> c | d

Counter({'a': 3, 'b': 2})