**Оглавление**

[15) функции высшего порядка. Частичное применение 1](#__RefHeading___Toc343_3745495915)

[16) структра проекта на Python. Модули. Иерархия классов. 2](#__RefHeading___Toc345_3745495915)

[17) модульное тестирование 3](#__RefHeading___Toc347_3745495915)

[18) параллельное программирование на Python. GIL 5](#__RefHeading___Toc349_3745495915)

[19) параллельное программирование на Python. Multiprocessing 7](#__RefHeading___Toc351_3745495915)

[20) параллельное программирование на Python. Гнераторы 9](#__RefHeading___Toc353_3745495915)

[21) параллельное программирование на Python. Событийно-ориентированное прогамирование 10](#__RefHeading___Toc355_3745495915)

[22) Применение Python в реальной жизни. Научные вычисления 11](#__RefHeading___Toc357_3745495915)

[23) Веб программирование Python 12](#__RefHeading___Toc359_3745495915)

[24) Искусственный интеллект на Python 13](#__RefHeading___Toc361_3745495915)

# 15) функции высшего порядка. Частичное применение

Функции высшего порядка могут принимать функции как аргументы и возвращать их.

def summation(nums):

return sum(nums)

def action(func, numbers):

return func(numbers)

print(action(summation, [1, 2, 3]))

# Output is 6 # Вывод 6

Частичное применение позволяет вызвать функцию без использования всех заданных аргументов

def power(base, exponent):

return base \*\* exponent

----

from functools import partial

square = partial(power, exponent=2)

print(square(2))

# output is 4 # вывод 4

powers = []

for x in range(2, 1001):

powers.append(partial(power, exponent = x))

print(powers[0](3))

# output is 9 # вывод 9

# 16) структра проекта на Python. Модули. Иерархия классов.

Объектно-ориентированная парадигма программирования (и, в частности, принцип наследования)

подразумевает построение иерархии классов, которая бы в совокупности наиболее эффективным образом решала поставленную задачу.

В Python все классы наследуются от класса object.

Встроенные классы Python также имеют свою иерархию, например, для типа bool она выглядит как:

object -> int -> bool

# 17) модульное тестирование

неплохая статья : https://habr.com/ru/company/mailru/blog/418929/

наш пример: https://github.com/wKenpachw/MorozovAP/blob/master/fourthlab/atribute\_test.py

Цели модульного тестирования:

1)регрессия: поправить что-нибудь в коде, запустить тесты и узнать, что ничего не сломалось

2)оценить влияние архитектуры  
 Если в проекте вы вводите обязательное модульное тестирование, или просто договариваетесь с разработчиками о применении модульных тестов, это немедленно отразится на стиле написания кода  
 Невозможно писать функции на 300 строк с 50 локальными переменными и 15 параметрами, если эти функции будут подвергаться модульному тестированию  
 Кроме того, благодаря этим тестам станут понятнее интерфейсы и проявятся какие-то проблемные места  
 Ведь если код не ахти, то и тест будет кривой, и это сразу бросится в глаза

3)сделать код понятнее  
 Допустим, вы пришли в новый проект и вам дали 50 Мб исходников  
 Возможно, вы просто не сможете в них разобраться  
 Если модульных тестов нет, то единственный способ познакомиться с работой кода, помимо чтения исходников, это «метод тыка»  
 Но если система достаточно сложная, то чтобы добраться через интерфейс до нужных кусков кода, может понадобиться много времени  
 А благодаря модульным тестам вы можете посмотреть, как код исполняется из любого места.

4)упростить отладку. К примеру, вы нашли какой-то класс и хотите его отладить. Если вместо модульных тестов есть только системные, или вообще никаких тестов нет, то остается только добираться до нужного места через интерфейс.

5)комфорт

Могу взять незнакомый исходник, поправить три строки, запустить тесты и убедиться, что код работает, как задумано. И речь даже не о том, чтобы тесты были зелеными: они могут быть и красными, но именно там, где я ожидаю. То есть я понимаю, как работает код.

Реализация на Python:

Мы используем стандартную библиотеку unittest из семейства xUnit.

Модульный тест — это обычный код, которому присуща некая стандартная архитектура. Все модульные тесты состоят из трех этапов: setup, exercise и verify. Вы подготавливаете данные, запускаете тесты и смотрите, всё ли пришло в нужное состояние.

Setup:

Самый сложный и интересный этап. Привести систему в изначальное состояние, из которого вы хотите её тестировать, может быть очень непросто. А состояние системы может быть сколь угодно сложным.

exercise:

Про этот этап рассказывать особо нечего. Единственное, что здесь может пойти не так, — это обращение вовне, например, в интернет.

Verify:

На этом этапе мы активно используем самописные assert’ы, даже однострочные

unittest:

пишем какой-то класс, функцию, что угоно, что хотим проверить

далее создаем файл для тестирования, импортируем написанный код и создаем класс сеттирования, унаследованный от unittest.TestCase

в нем должен вызываем методы для тестирования

1)assertEqual - метод сравненивает то, что мы хотим получить с тем, что мы получаем используя наш первый класс:

self.assertEqual(result(то, что получим используя наш первый класс, value (ожидаемое значение)

при совпадении результатов тест будет пройден успешно

2) assertNotEqual - тоже самое, но успешно, когда значения расходятся

3) assertTrue и assertFalse - когда ожидаем True или False

4)assertIs и assertIsNot - сравниваем два результата (скорее всего две сущности, а не значения) и сравниваем их

5) assertIsNone и assertIsNotNone - получаем None или не None

их ещё много всяких, но больше мы точно не проходили.

# 18) параллельное программирование на Python. GIL

https://tproger.ru/translations/global-interpreter-lock-guide/

GIL – Global Interpreter Lock

это своеобразная блокировка, позволяющая только одному потоку управлять интерпретатором Python. Это означает, что в любой момент времени будет выполняться только один конкретный поток.

Он добавляет правило: любое выполнение байткода в Python требует блокировки интерпретатора. В таком случае можно исключить взаимоблокировку, т. к. GIL будет единственной блокировкой в приложении. К тому же его влияние на производительность процессора совсем не критично. Однако стоит помнить, что GIL уверенно делает любую программу однопоточной.

Ниже приведена простая CPU-bound программа, которая попросту ведёт обратный отсчёт:

# single\_threaded.py

import time

from threading import Thread

COUNT = 50000000

def countdown(n):

while n > 0:

n -= 1

start = time.time()

countdown(COUNT)

end = time.time()

print('Затраченное время -', end - start)

Ниже приведена та же программа, с небольшим изменением. Теперь обратный отсчёт ведётся в двух параллельных потоках:

# multi\_threaded.py

import time

from threading import Thread

COUNT = 50000000

def countdown(n):

while n > 0:

n -= 1

t1 = Thread(target=countdown, args=(COUNT//2,))

t2 = Thread(target=countdown, args=(COUNT//2,))

start = time.time()

t1.start()

t2.start()

t1.join()

t2.join()

end = time.time()

print('Затраченное время -', end - start)

Многопроцессность против многопоточности. Довольно популярное решение, поскольку у каждого Python-процесса есть собственный интерпретатор с выделенной под него памятью, поэтому с GIL проблем не будет. В Python уже есть модуль multiprocessing, который упрощает создание процессов к такому виду:

from multiprocessing import Pool

import time

COUNT = 50000000

def countdown(n):

while n > 0:

n -= 1

if \_\_name\_\_ == '\_\_main\_\_':

pool = Pool(processes=2)

start = time.time()

r1 = pool.apply\_async(countdown, [COUNT//2])

r2 = pool.apply\_async(countdown, [COUNT//2])

pool.close()

pool.join()

end = time.time()

print('Затраченное время в секундах -', end - start)

# 19) параллельное программирование на Python. Multiprocessing

https://geekbrains.ru/posts/python\_threading\_part1

это библиотека для управления процессами

Создаем потоки:

Метод 1 — «функциональный»

Для работы с потоками из модуля threading импортируем класс Thread. В начале кода пишем:

from threading import Thread

После этого нам будет доступна функция Thread() — с ней легко создавать потоки. Синтаксис такой:

variable = Thread(target=function\_name, args=(arg1, arg2,))

Первый параметр target — это «целевая» функция, которая определяет поведение потока и создаётся заранее. Следом идёт список аргументов. Если судьбу аргументов (например, кто будет делимым, а кто делителем в уравнении) определяет их позиция, их записывают как args=(x,y). Если же вам нужны аргументы в виде пар «ключ-значение», используйте запись вида kwargs={‘prop’:120}.

Ради удобства отладки можно также дать новому потоку имя. Для этого среди параметров функции прописывают name=«Имя потока». По умолчанию name хранит значение null. А ещё потоки можно группировать с помощью параметра group, который по умолчанию — None.

За дело! Пусть два потока параллельно выводят каждый в свой файл заданное число строк. Для начала нам понадобится функция, которая выполнит задуманный нами сценарий. Аргументами целевой функции будут число строк и имя текстового файла для записи.

Давайте попробуем:

#coding: UTF-8

from threading import Thread

def prescript(thefile, num):

with open(thefile, 'w') as f:

for i in range(num):

if num > 500:

f.write('МногоБукв\n')

else:

f.write('МалоБукв\n')

thread1 = Thread(target=prescript, args=('f1.txt', 200,))

thread2 = Thread(target=prescript, args=('f2.txt', 1000,))

thread1.start()

thread2.start()

thread1.join()

thread2.join()

start() запускает ранее созданный поток

Метод join() останавливает поток, когда тот выполнит свои задачи. В качестве параметра в скобках можно указать, на сколько секунд блокировать поток перед продолжением его работы.

Для потока со сложным поведением обычно пишут отдельный класс, который наследуют от Thread из модуля threading. В этом случае программу действий потока прописывают в методе run() созданного класса.

import threading

class MyThread(threading.Thread):

def \_\_init\_\_(self, num):

super().\_\_init\_\_(self, name="threddy" + num)

self.num = num

def run(self):

print ("Thread ", self.num),

thread1 = MyThread("1")

thread2 = MyThread("2")

thread1.start()

thread2.start()

thread1.join()

thread2.join()

Чтобы управлять потоками, нужно следить, как они себя ведут. И для этого в threading есть специальные методы:

current\_thread() — смотрим, какой поток вызвал функцию;

active\_count() — считаем работающие в данный момент экземпляры класса Thread;

enumerate() — получаем список работающих потоков.

Ещё можно управлять потоком через методы класса:

is\_alive() — спрашиваем поток: «Жив ещё?» — получаем true или false;

getName() — узнаём имя потока;

setName(any\_name) — даём потоку имя;

# 20) параллельное программирование на Python. Гнераторы

https://webdevblog.ru/vvedenie-v-generatory-python/

Генераторы — итерируемые объекты, но, в общем случае, вы можете их использовать только один раз. Это связано с тем, что они не хранят все значения в памяти, а генерируют значения «на лету» — по мере запроса

Yield — это ключевое слово которое используется так же, как и слово return. Разница в том, что функция при этом начинает возвращать генератор вместо значения.

когда вы вызываете функцию, в теле которой находится yield, выполнение этой функции не происходит. Вместо выполнения, функция вернёт объект-генератор.

При первом исполнении кода тела функции код будет выполнен с начала и до первого встретившегося оператора yield. После этого будет возвращено первое значение и выполнение тела функции опять приостановлено. Запрос следующего значения у генератора во время итерации заставит код тела функции выполняться дальше (с предыдущего yield’а), пока не встретится следующий yield. Генератор считается «закончившимся» в случае если при очередном исполнении кода тела функции не было встречено ни одного оператора yield.

def countdown(num):

print('Starting')

while num > 0:

yield num

num -= 1

Объекты генератора выполняются при вызове next(): next(val)

Пример 1

def emit\_lines(pattern=None):

lines = []

for dir\_path, dir\_names, file\_names in os.walk('test/'):

for file\_name in file\_names:

if file\_name.endswith('.py'):

for line in open(os.path.join(dir\_path, file\_name)):

if pattern in line:

lines.append(line)

return lines

Эта функция просматривает набор файлов в указанном каталоге. Она открывает каждый файл и затем просматривает каждую строку, чтобы проверить соответствие шаблону.

# 21) параллельное программирование на Python. Событийно-ориентированное прогамирование

смотреть пример в папрке calc и https://python-scripts.com/pyqt5-calculator

Событийно-ориентированное программирование (event-driven programming) - это парадигма программирования, в которой выполнение программы определяется событиями — действиями пользователя (клавиатура, мышь), сообщениями других программ и потоков, событиями операционной системы (например, поступлением сетевого пакета).

Событийно-ориентированное программирование можно также определить как способ построения компьютерной программы, при котором в коде (как правило, в головной функции программы) явным образом выделяется главный цикл приложения, тело которого состоит из двух частей: выборки события и обработки события.

Событийно-ориентированное программирование, как правило, применяется в трех случаях:

при построении пользовательских интерфейсов (в том числе графических);

при создании серверных приложений в случае, если по тем или иным причинам нежелательно порождение обслуживающих процессов;

при программировании игр, в которых осуществляется управление множеством объектов.

# 22) Применение Python в реальной жизни. Научные вычисления

# 23) Веб программирование Python

https://ru.wikibooks.org/wiki/Flask - тут рассказываются основы того, как и что делать

для ответа на этот билет будем рассказывать о Flask и пытаться что-то показать на примере нашей лабы в папке site

Flask является микрофреймворком Python, который имеет модульный дизайн. Данный фреймворк предназначен для создания веб-приложений.

Платформа поддерживает маршрутизацию URL-адресов, шаблоны (с Jinja2), управление сеансами и имеет некоторые фишки в области безопасности.

from flask import Flask

app = Flask(\_\_name\_\_)

@app.route('/')

def hello\_world():

return 'Hello World!'

if \_\_name\_\_ == '\_\_main\_\_':

app.run()

Рассмотрим, как работает программа, которую вы запустили выше:

Сначала мы импортируем Flask класс. Экземпляр этого класса будет нашим WSGI приложением. Первым аргументом является имя модуля приложения. Если вы используете один модуль (как в данном примере), вы должны использовать \_\_name\_\_, потому что в зависимости от того, было ли это начато как приложение или как импорт модуля, название будет другим ('\_\_main\_\_' по сравнению с реальным именем импорта).

Далее мы создаем экземпляр этого класса. Мы передаем ему имя модуля или пакета. Это необходимо, так как Flask не знает, где искать шаблоны, статические файлы, и так далее.

Затем мы используем route(). Декоратор говорит Flask, что URL должен вызывать нашу функцию.

Функция задает имя, которое также используется для создания URL-адресов для этой функции, и возвращает сообщение, что мы хотим отобразить в браузере пользователя.

Наконец, мы используем run() функцию для запуска локального сервера с нашим приложением. Условие \_\_name\_\_ == «\_\_main\_\_» означает, что сервер работает только в том случае, если скрипт выполняется непосредственно из Python интерпретатора и не используется в качестве импортированного модуля.

# 24) Искусственный интеллект на Python