

Лекция 12

Язык программирования Python.

Хайрулин Сергей Сергеевич

email: s.khairulin@g.nsu.ru, s.khayrulin@gmail.com

Ссылка на [материалы](#)

План

- Лекции/практические занятия
 - Тест
- Дифференцированный зачет в конце семестра
 - Защита задания

Литература

Начальный уровень

- Mark Pilgrim. Dive into Python - <http://www.diveintopython.net/>
- Марк Лутц. Изучаем Python, 4-е издание // Символ-Плюс 2011.
- ...

Стандарт/Документация

- PEP-8 - <https://www.python.org/dev/peps/pep-0008/>
- <https://www.python.org/>
- <https://github.com/python/cpython>

Экспертный уровень

- Лучано Рамальо: Python. К вершинам мастерства
- Mitchell L. Model. Bioinformatics Programming Using Python // O'Reilly 2010.

Версии Python

- Python 2 вышел 2010 году последняя версия 2.7.16 - исправлялись только баги(ошибки) с января 2020 года поддержка прекращена.
- Python 3 появился в 2008, является актуальной версией языка.
Текущая стабильная версия 3.9, в разработке 3.10
 - Python 3 не гарантирует совместимости кода с Python 2

Summary

- Декораторы
- Typing
- Параллельное программирование в Python
 - GIL
- Библиотеки
 - анализ данных
 - визуализация данных

Декораторы

Специализированные языковые конструкции, позволяющие оборачивать функции. По сути декоратор это функция, которая возвращает функцию как результат своей работы.

Декораторы

Декораторы выделяются символом @

```
def dec_name(func):  
    def foo(...):  
        ...  
        func(...)  
        ...  
    return foo
```

```
@dec_name  
def func_name(...):  
    ...
```


Декораторы

Конструкция

```
@decorator_name  
def func():  
    ...
```

Эквивалентна вызову функции `decorator_name` с аргументом `func`

```
decorator_name(func)
```

Декораторы

[illegible]

Typing

Появились в Python с версии 3.4. Возможность явно указывать типы данных на которые будет ссылаться переменная. Это не накладывает абсолютно никаких ограничений на динамическую типизацию, **НО** дает возможность как программисту так и статическому анализатору кода, предупреждать о несоответствии типов в случае если они встречаются.

Typing

Для того чтобы воспользоваться Тайпингами нужно импортировать пакет typing. Этот пакет содержит определение всех базовых типов.

```
12 > tp.py > ...  
1  |from typing import Dict, List, Any, Tuple
```

Typing

```
3
4 import typing as t
5
6 class A:
7     pass
8
9 def func(arg1: int, arg2: t.ByteString, arg3: t.List[str], arg4: t.Dict[str, t.Any]) → bool:
10     s: A = A()
11     return True
```

Параллельное программирование в Python

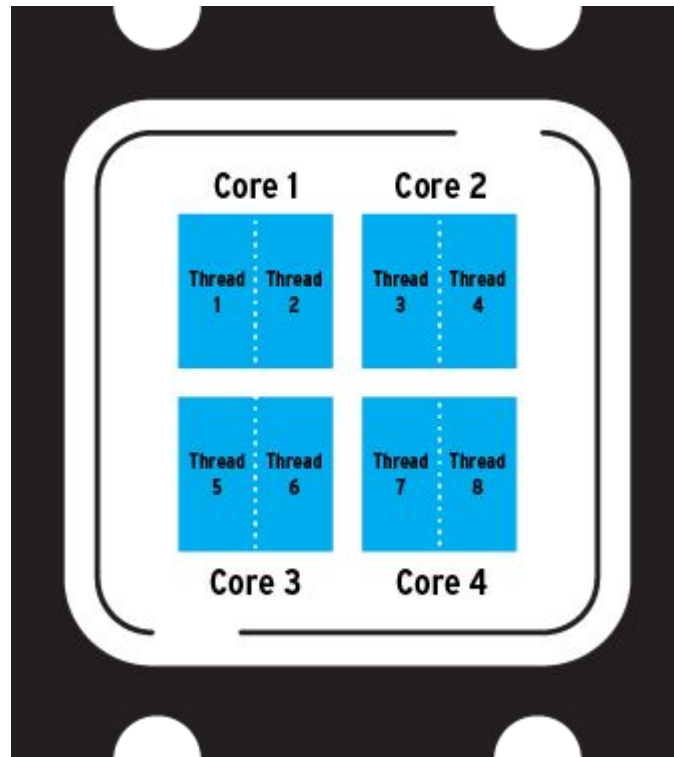
Параллельное программирование (параллельные вычисления) - парадигма программирования, которая подразумевает параллельную (одновременную) обработку данных в несколько независимых потоков.

Параллельное программирование в Python

В Python параллельные вычисления организованы за счет использование пакета [threading](#) и [multiprocessing](#)

Потоки (ЦПУ)

На уровне процессора потоком исполнения команд можно называть отдельное независимое ядро процессора. Это верно только относительно многоядерных ЦПУ.

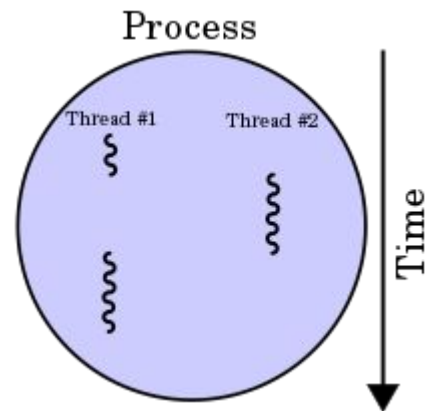


Потоки (OS)

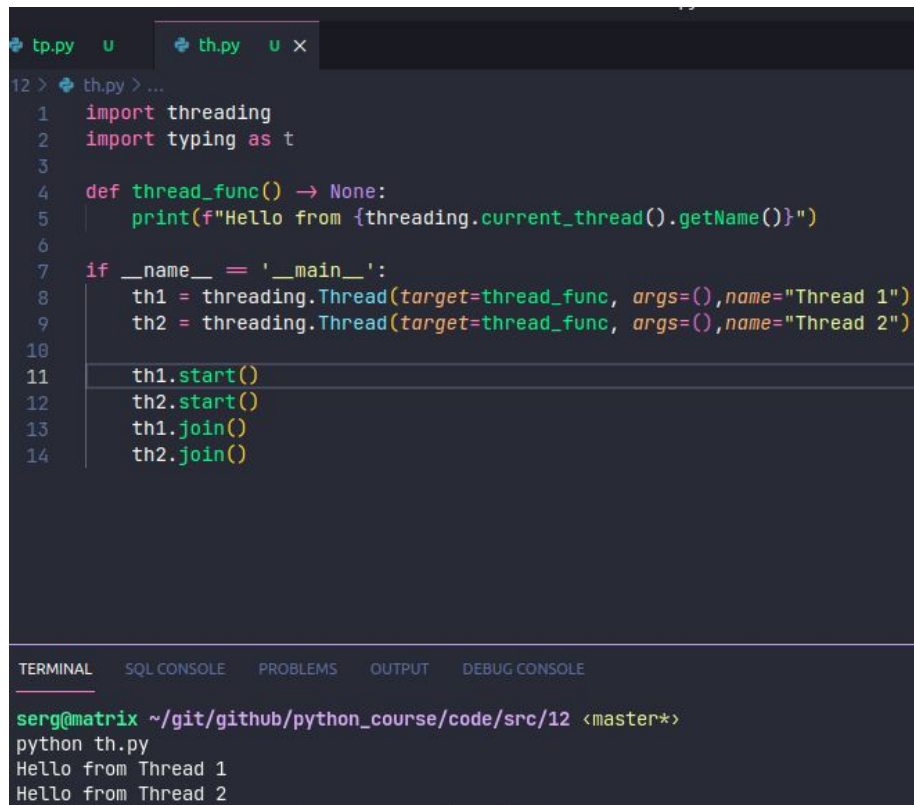
Относительно ядра операционной системы, все программы запускаются в отдельном процессе (Process), который однако может иметь как несколько дочерних процессов (подпроцесс), так и несколько потоков (Thread), которые выполняются “параллельно”.

- Количество потоков может сильно отличаться от количества ядер процессора.
Как такое возможно

Все дело в том, что потоки OS не тоже самое что ядро процессора. Жизненный цикл потока регулируется планировщиком задач OS



Поток Python



The image shows a code editor with a dark theme. At the top, there are two tabs: 'tp.py' and 'th.py'. The 'th.py' tab is active, displaying a Python script. The script imports 'threading' and 'typing as t'. It defines a function 'thread_func()' that prints a message indicating the current thread's name. In the main block, two threads are created: 'th1' and 'th2', both targeting 'thread_func'. They are started with 'start()' and then joined with 'join()' to wait for completion. Below the code editor is a terminal window with tabs for 'TERMINAL', 'SQL CONSOLE', 'PROBLEMS', 'OUTPUT', and 'DEBUG CONSOLE'. The 'TERMINAL' tab is active, showing the command 'python th.py' and its output: 'Hello from Thread 1' and 'Hello from Thread 2'.

```
12 > th.py > ...
1  import threading
2  import typing as t
3
4  def thread_func() -> None:
5      print(f"Hello from {threading.current_thread().getName()}")
6
7  if __name__ == '__main__':
8      th1 = threading.Thread(target=thread_func, args=(), name="Thread 1")
9      th2 = threading.Thread(target=thread_func, args=(), name="Thread 2")
10
11     th1.start()
12     th2.start()
13     th1.join()
14     th2.join()
```

TERMINAL SQL CONSOLE PROBLEMS OUTPUT DEBUG CONSOLE

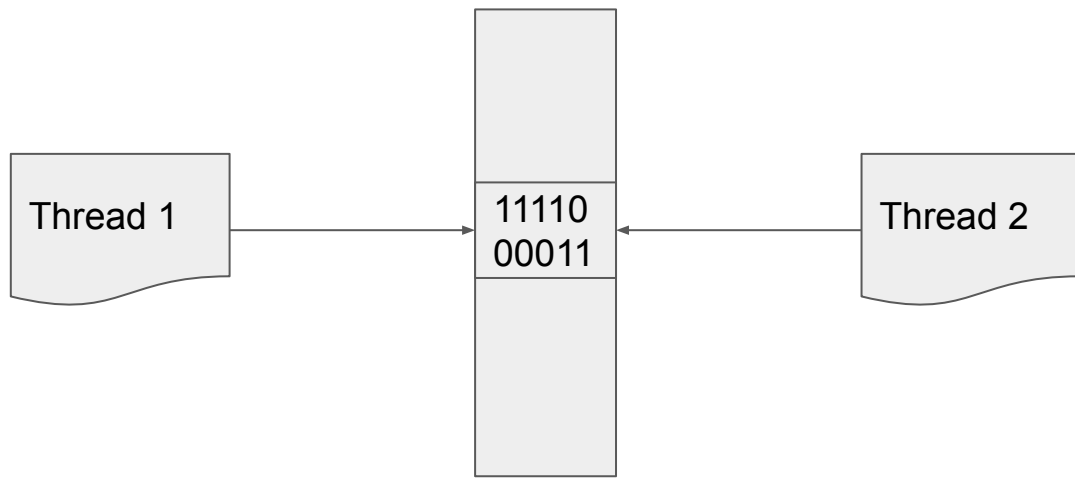
serg@matrix ~/git/github/python_course/code/src/12 <master*>
python th.py
Hello from Thread 1
Hello from Thread 2

Примитивы ПП

- Потоки
- Примитивы синхронизации
 - блокировки/мьютексы
 - семафор
 - события

Критическая секция

Критической секцией программы называют то место программы, в котором потенциально несколько потоков могут менять одни и те же данные.



Синхронизация и состояние гонки

В случае если два или несколько выполняющихся потоков одновременно обращаются к одному и тому же участку памяти как на чтение/запись, можно столкнуться с неопределенным поведением программы. Чтобы предотвратить такие ситуации используются специальные примитивы такие как блокировки, мьютексы, семафоры.

Блокировки

Блокировка - предотвращение одновременного выполнения одного куска кода несколькими потоками. Получается что блокировка гарантирует вам, что только один поток выполняет работу после блокировки, другие выстраиваются в очередь и ждут, когда блокировка освободиться для захвата.

Блокировки

```
12 > th2.py > ...
1  import threading
2  import time
3  import typing as t
4
5  COUNTER = 0
6
7  def thread_func(by: int) -> None:
8      global COUNTER
9      local_counter = COUNTER
10     local_counter += by
11     time.sleep(0.1)
12
13     COUNTER = local_counter
14     print(f"{COUNTER=}")
15
16
17 if __name__ == '__main__':
18     th1 = threading.Thread(target=thread_func, args=(10,), name="Thread 1")
19     th2 = threading.Thread(target=thread_func, args=(20,), name="Thread 2")
20
21     th1.start()
22     th2.start()
23     th1.join()
24     th2.join()
25     print(COUNTER)
```

TERMINAL SQL CONSOLE PROBLEMS OUTPUT DEBUG CONSOLE

```
serg@matrix ~/git/github/python_course/code/src/12 <master*>
python th2.py
COUNTER=10
COUNTER=20
20
```

Блокировки

```
12 > th2_lock.py > ...
1 import threading
2 import time
3 import typing as t
4
5 COUNTER = 0
6
7 def thread_func(by: int, l: threading.Lock) -> None:
8     global COUNTER
9     l.acquire()
10    local_counter = COUNTER
11    local_counter += by
12    time.sleep(0.1)
13
14    COUNTER = local_counter
15    print(f"{COUNTER=}")
16    l.release()
17
18
19
20 if __name__ == '__main__':
21     lock = threading.Lock()
22     th1 = threading.Thread(target=thread_func, args=(10, lock), name="Thread 1")
23     th2 = threading.Thread(target=thread_func, args=(20, lock), name="Thread 2")
24
25     th1.start()
26     th2.start()
27     th1.join()
28     th2.join()
29     print(COUNTER)
```

TERMINAL SQL CONSOLE PROBLEMS OUTPUT DEBUG CONSOLE

```
serg@matrix ~/git/github/python_course/code/src/12 <master>
python th2_lock.py
COUNTER=10
COUNTER=30
30
```


Процесс

Процесс - более ресурсоемкая сущность в сравнение с потоком, обладает своим адресным пространством и не может разделять его. Работает независимо от других процессов, регулируется только планировщиком задач операционной системы.

Процесс

```
12 > ps.py > ...
1  import multiprocessing
2  import typing as t
3
4  COUNTER = 1
5
6  def process_func() → None:
7      global COUNTER
8      COUNTER += 1
9      print(f"{COUNTER=} in process {multiprocessing.current_process().name}")
10
11 if __name__ == '__main__':
12     p1 = multiprocessing.Process(target=process_func, name="Process 1")
13     p2 = multiprocessing.Process(target=process_func, name="Process 2")
14
15     p1.start()
16     p2.start()
17
18     p1.join()
19     p2.join()
20
```

TERMINAL SQL CONSOLE PROBLEMS OUTPUT DEBUG CONSOLE

```
serg@matrix ~/git/github/python_course/code/src/12 <master*>
python ps.py
COUNTER=2 in process Process 1
COUNTER=2 in process Process 2
```

GIL

- Что такое GIL?
- Зачем он нужен?

GIL

GIL - global interpreter lock, глобальная объект предотвращающий одновременное исполнение кода в двух или нескольких потоках. То есть в каждый момент времени выполняется только один поток.

GIL



GIL

Зачем он нужен!?

- Проблема с системой управления памятью

Несколько потоков могут менять состояние объекта одновременно, что может привести к некорректному поведению интерпретатора.

CPU-bound операции

Вычислительные задачи требующие активного использования CPU - называются CPU-bound. Например

- умножение матриц
- обработка изображений
- ...

На такие операции, при условии, что они могут быть распараллелены очень сильно влияет GIL

CPU-bound операции

```
1  import threading
2  import time
3
4  COUNT = 50000000
5
6  def countdown(n):
7      while n > 0:
8          n -= 1
9
10 if __name__ == '__main__':
11     start = time.time()
12     countdown(COUNT)
13     end = time.time()
14     print(f"single thread time {end - start}")
15
16     th1 = threading.Thread(target=countdown, args=(COUNT//2,))
17     th2 = threading.Thread(target=countdown, args=(COUNT//2,))
18     start = time.time()
19     th1.start()
20     th2.start()
21     th1.join()
22     th2.join()
23     end = time.time()
24     print(f"two thread time {end - start}")
```

TERMINAL

SQL CONSOLE

PROBLEMS

OUTPUT

DEBUG CONSOLE

serg@matrix ~/git/github/python_course/code/src/12 <master*>

python gil_cpu.py

single thread time 2.3090739250183105

two thread time 3.452542304992676

IO-bound операции

Операции требующие ввода вывода данных из сторонних источников называются IO-bound

- сетевое общение с источниками данных
- общение с другими системами OS

Библиотеки

- ml/ds
 - [scikit-learn](#)
 - [keras](#)
 - ...
- science
 - [scipy](#)
 - [numpy](#)
 - ...
- визуализация данных
 - [matplotlib](#)
 - [pandas](#)
 - ...

Спасибо за внимание!

Lab 4

Написать калькулятор выполняющий команды арифметических операций используя классы и стандартные библиотеки python. Калькулятор должен уметь исполнять команды, выводить ошибки о неверных значениях операндов, необъявленных переменных, неизвестных командах итп.

Lab 4 (на 3)

Реализовать простой набор арифметических команд (сложение, вычитание, умножение, деление) с целыми числами и числами с плавающей точкой:

- `ADD operand1 operand2 --- operand1 + operand2`
- `SUB operand1 operand2 --- operand1 - operand2`
- `MUL operand1 operand2 --- operand1 * operand2`
- `DIV operand1 operand2 --- operand1 / operand2`

Lab 4 (на 4)

3 + Реализовать работу с переменными (перед работой с переменной в калькуляторе её обязательно нужно объявлять, если переменная не объявлена, то калькулятор должен сообщить об этом в строке вывода)!

- `SET var_name var_value --- var_name = var_value` - объявление переменной и присваивание ей определенного значения
- `PRINT var_name --- print(var_name)` - вывод значения переменной
- в базовом наборе команд могут использоваться переменные, причем
 - если переменная в правом операнде, то используется ее значение:

```
>>> SET x 5
>>> ADD 2 x    // ADD 2 5
7
```

- если переменная в левом операнде, то используется её значение и результат выражения присваивается этой переменной:

```
>>> SET x 5
>>> ADD x 2    // x += 2
>>> MUL x 2    // x *= 2
>>> PRINT x
14
```

Lab 4 (на 5)

4 + Реализовать работу с функциями.

- `DEF func_name : arg_1 ... arg_n : command_1 operand_1 operand_2 ; ... ; RETURN var_name_or_value ; ---`
объявление функции с набором аргументов, телом функции (набор команд) и возвращаемым значением
- `CALL func_name arg_val_1 ... arg_val_n ---` вызов функции по имени с набором аргументов
- `CALL func_name arg_val_1 ... arg_val_n INTO var_name ---` вызов функции и присвоение результата переменной

Пример функции возведения во вторую степень:

```
>>> DEF pow2 : x : MUL x x ; RETURN x ;
>>> CALL pow2 3
9
>>> SET x 5
>>> CALL pow2 x INTO x
>>> PRINT x
25
```