

Лекция 3

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 3.8.5 -> в пред релиз 3.9, в разработке 3.10
 - Python 3 не гарантирует совместимости кода с Python 2

План занятия

- Как написать и запустить программу
- Алгоритмы
- Программные блоки
- Логические операторы
- Циклы
 - **while**
 - **for** итерирование над объектами
- Условные операторы
 - **if**
 - **if ... else**
 - **if ... elif ...**
- Практика

Динамическая типизация

***Strong** typing means that the type of a value doesn't change in unexpected ways. A string containing only digits doesn't magically become a number, as may happen in Perl. Every change of type requires an explicit conversion.*

***Dynamic** typing means that runtime objects (values) have a type, as opposed to static typing where variables have a type.*

Python - **динамически** типизированный язык.

При динамической типизации переменная не знает тип значение на которое в данный момент эта переменная указывает (**ЭТО ВАЖНО**) - при этом само значение хранит эту информацию.

Динамическая типизация

Не стоит считать, что все языки поддерживают динамическую типизацию, как и по типу выполнения кода языки, также можно объединять по типу типизации.

Dynamic typing Language

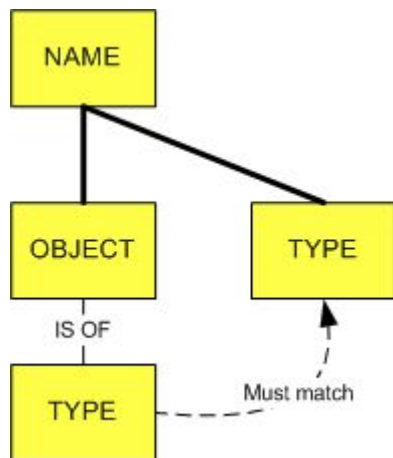
- ...
- Python
- JS
- ...

Static typing Language

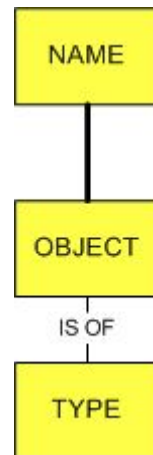
- ...
- C++
- C
- ...

Динамическая типизация

Статическая типизация



Динамическая типизация



Динамическая типизация (+/-)

+	-
Удобство объявления	Неочевидное поведение программы
Красиво и компактно	Memory Overhead

Динамическая типизация in action (Python)

Python (динамическая типизация все что написано ниже верно и не вызовет ошибок)

```
x = 10
```

```
x = 'Hello World'
```

```
x = 3.14
```

C++ (статическая типизация)

```
int intVar = 10;
```

```
string stringVar = "Hello world"
```

```
float floatVar = 3.14
```

```
// а вот так в C++ сделать не получится компилятор станет ругаться
```

```
floatVar = "Hello world"
```

Переменные(базовые операции)

Для сокращения записи, если выражение

подразумевает изменение той же переменной,

то разумно использовать следующие варианты записи

`x += 1` *# тоже самое что и $x = x + 1$*

`x -= 1` *# тоже самое что и $x = x - 1$*

`x *= 1` *# тоже самой что и $x = x * 1$*

`x /= 2` *# тоже самое что и $x = x / 2$*

`x **= 2` *# тоже самое что и $x = x ** 2$*

Присвоение

`x = 1`

`y = 2`

`x = y` *# теперь переменная x равна 2*

`z = x + y` *# теперь переменная z равна 4*

Переменные (базовые операции)

Узнать тип переменной можно с помощью функции `type(...)`

```
x = 10
```

```
# результат выполнения этой функции будет
```

```
# тип объекта переданного в аргументе
```

```
# в нашем случае int
```

```
type(x)
```

Узнать месторасположение объекта в памяти можно с помощью `id(...)`

```
# результат зависит от платформы
```

```
# НО точно будет являться целым числом
```

```
id(x)
```

Работа с вводом/выводом

Взаимодействие с пользователем может настраиваться через стандартные потоки ввода и вывода `stdin/stdout`.

Для получения входной информации от пользователя можно воспользоваться вызовом функции [`input\(...\)`](#)

Для вывода информации из программы можно воспользоваться функцией [`print\(...\)`](#)

Важно: функция `input` блокирует выполнение скрипта и ждет ввода пользователя, для того чтобы продолжить работу. Кроме того сама функция возвращает введенную строку, **НЕ ЗАБЫВАЙТЕ КОНВЕРТИРОВАТЬ ТИПЫ**

Работа с вводом/выводом

```
In [21]: my_name = input("Put your name here: ")  
Put your name here: Sergey
```

```
In [22]: print(my_name)  
Sergey
```

```
In [23]: █
```

Что такое программа?

Комбинация компьютерных инструкций и данных, позволяющая аппаратному обеспечению вычислительной системы выполнять вычисления или функции управления (стандарт ISO/IEC/IEEE 24765:2010).

Что такое программа (в терминах Python)

Набор исходного папок/файлов (пакетов/модулей) с исходным кодом. Каждый модуль содержит набор команд для интерпретатора выполнение которых начинается с точки входа, обычно мы явно указываем интерпретатору какой модуль(файл) является точкой входа для программы. Разберемся на примере...

Как написать и запустить программу

```
[17:03:04] serg :: serg-pc → ~/tmp»  
vim my_first_script.py
```

```
print("Hello Python!")
```

```
~  
~  
~  
~  
~  
~  
~
```

Запуск программы в UNIX-like системах

```
[17:06:19] serg :: serg-pc → ~/tmp»  
python my_first_script.py  
Hello Python!
```

```
my_first_script.py  
1 #!/usr/bin/python  
1  
2 print("Hello World")
```

```
sergey@CM003 ~/tmp via 🐘 v2.7.18 took 1m  
→ chmod +x my_first_script.py
```

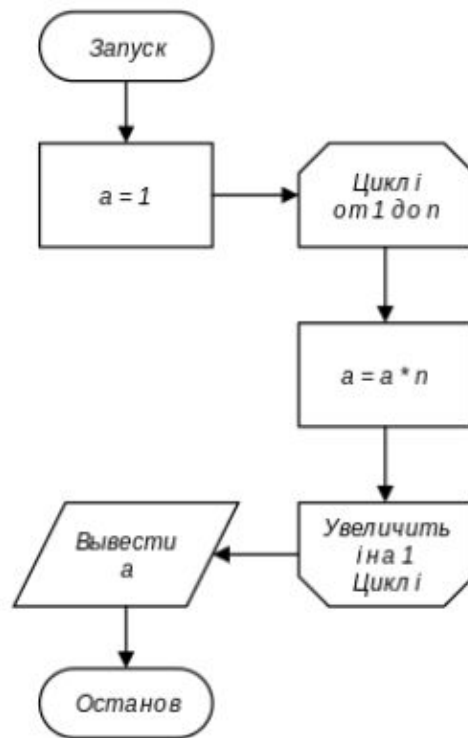
```
sergey@CM003 ~/tmp via 🐘 v2.7.18  
→ ./my_first_script.py  
Hello World
```

Алгоритмы

Алгоритм — набор инструкций, описывающих порядок действий исполнителя для достижения некоторого результата. Независимые инструкции могут выполняться в произвольном порядке, параллельно, если это позволяют используемые исполнители.

(Википедия)

Алгоритмы



Свойства

- Конечность описания
- Дискретность
- Направленность
- Массовость
- Детерминированность

Конечность описания

Конечность описания — любой алгоритм задается как набор инструкций конечных размеров, т. е. программа имеет конечную длину.

Дискретность

Дискретность — алгоритм выполняется по шагам, происходящим в дискретном времени. Шаги четко отделены друг от друга. В алгоритмах нельзя использовать аналоговые устройства и непрерывные методы.

Направленность

Направленность — у алгоритма есть входные и выходные данные. В алгоритме четко указывается, когда он останавливается, и что выдается на выходе после остановки.

Массовость

Массовость — алгоритм применим к некоторому достаточно большому классу однотипных задач, т. е. входные данные выбираются из некоторого, как правило, бесконечного множества.

Детерминированность

Детерминированность (или конечная недетерминированность) — вычисления продвигаются вперед детерминировано, т. е. вычислитель однозначно представляет, какие инструкции необходимо выполнить в текущий момент. Нельзя использовать случайные числа или методы. Конечная недетерминированность означает, что иногда в процессе работы алгоритма возникает несколько вариантов для дальнейшего хода вычислений, но таких вариантов лишь конечное.

Алгоритм вычисления чисел Фибоначчи

```
function Fibo(n)
    if n = 1 or n = 2
        return 1
    endif
    return Fibo(n - 1) + Fibo(n - 2)
endfunction
```

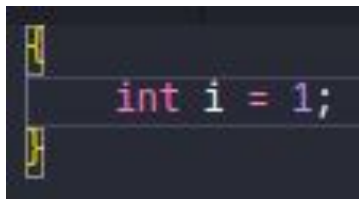
$$F_1 = 1, F_2 = 1, \dots, F_n = F_{n-1} + F_{n-2}$$

Алгоритм описан на псевдо языке. Реализация сделана на основе рекурсивного вызова функции Fibo(...). Что такое рекурсия мы выясним позже

Программные блоки

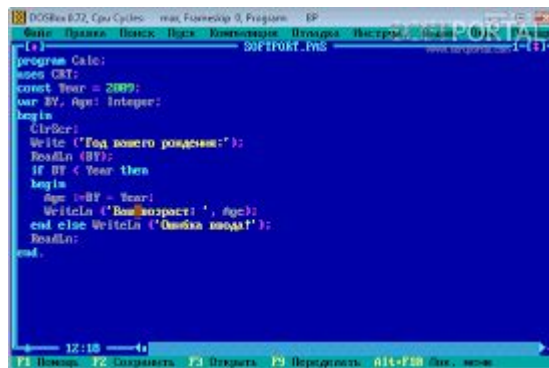
Куски программного кода, определяющие некоторую независимую часть логики. Обычно выделяются специализированными символами либо ключевыми словами

C/C++/Java/Go/C#...



```
int i = 1;
```

Pascal



```
program Calc;
uses CRT;
const Year = 2000;
var Y, Age: Integer;
begin
  ClrScr;
  Write ('Год моего рождения: ');
  ReadLn (Y);
  if Y < Year then
  begin
    Age := Y - Year;
    WriteLn ('Ваш возраст: ', Age);
  end else WriteLn ('Ошибка ввода!');
  ReadLn;
end.
```

Программные блоки

В python программные блоки выделяются 4мя пробелами или одной табуляцией - это позволяет визуально отделять блоки и дисциплинирует программиста писать более читаемы код.

```
def foo():  
    x = 1  
    x = x + 1  
    for i in range(10):  
        if i % 2 == 0:  
            print('I is even')  
        else:  
            print('I is odd')
```

```
def foo():  
    x = 1  
    x = x + 1|  
    for i in range(10):  
        if i % 2 == 0:  
            print('I is even')  
        else:  
            print('I is odd')
```

Программные блоки (примеры)

```
3  int main(int argv, char** argc)
4  {
5      int a = 1;
6      if (a == 1) {
7          std::cout << a << std::endl;
8      }
9      return 0;
10 }
```

```
def main(argv, argc):
    a = 1
    if a == 1:
        print(a)
    return 0
```

Операции сравнения

Операция сравнения может быть применена только к объектам, которые можно сравнивать между собой! Кроме того объекты должны поддерживать соответствующую операцию.

Вы не можете сравнивать, к примеру, число со строкой, **НО** вы можете проверять на равенство одного объекта с другим

Результатом сравнения всегда является значение типа **bool**, то есть **True** или **False**

Операции сравнения

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

https://www.tutorialspoint.com/python/python_basic_operators.htm

Операции сравнения

```
In [1]: 1 < 2
```

```
Out[1]: True
```

```
In [2]: 1 <= 1
```

```
Out[2]: True
```

```
In [3]: 1 >= 1
```

```
Out[3]: True
```

```
In [4]: 1 > 2
```

```
Out[4]: False
```

```
In [5]: 1 != 2
```

```
Out[5]: True
```

```
In [6]: 1 != 1
```

```
Out[6]: False
```

```
In [7]: 1 == 1
```

```
Out[7]: True
```

Логические операторы

Operator	Description	Example
and Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not(a and b) is false.

Оператор **is**

Возвращает значение **True**, если переменные по обе стороны от оператора указывают на один и тот же объект, и **False** в противном случае. **ТО ЕСТЬ СРАВНЕНИЕ ИДЕТ НЕ ПО ЗНАЧЕНИЮ НА КОТОРЫЕ УКАЗЫВАЮТ ПЕРЕМЕННЫЕ, А ПО АДРЕСУ НА КОТОРЫЙ ССЫЛАЮТСЯ ПЕРЕМЕННЫЕ.**

```
In [24]: a = 1
In [25]: b = 1
In [26]: a is b
Out[26]: True
```

Оператор **in**

Возвращает **True**, если находит переменную в указанной последовательности, и **False** в противном случае.

```
In [1]: l = [1,2,3]
```

```
In [2]: 1 in l
```

```
Out[2]: True
```

```
In [3]: x = 1
```

```
In [4]: x in l
```

```
Out[4]: True
```

```
In [5]: id(1)
```

```
Out[5]: 9666944
```

```
In [6]: id(l[0])
```

```
Out[6]: 9666944
```

```
In [7]: id(x)
```

```
Out[7]: 9666944
```

Оператор and

expression_1 and expression_2 -> True если оба выражения истинны в других случаях False

x	y	x and y
True	True	True
False	False	False
False	True	False
True	False	False

Оператор or

expression_1 or expression_2 -> True если **хотя бы одно** выражения истинно
в других случаях False

x	y	x and y
True	True	True
False	False	False
False	True	True
True	False	True

Оператор not

not expression_1 -> True если выражение False и наоборот

x	not x
True	False
False	True

Tricks

Выражения можно комбинировать они могут быть очень сложными, **НО** будьте осторожны с этим слишком сложные выражения сложно отлаживать и поддерживать, так что лучше их разбивать на мелкие составные части.

```
In [9]: x = True  
  
In [10]: y = False  
  
In [11]: x is True and not y or y is False and x is False  
Out[11]: True
```

Ветвление (программирование)

Часто нужно поведение программы может зависеть от данных над которыми она работает и бизнес логика программы может меняться. В подобных ситуациях полезно использовать логические ветвления. Python как и все тьюринг полные языки программирования поддерживают такие операции. Ветвление организуется с помощью операторов условного перехода **if**, **else**, **elif**.

Условные операторы if

Синтаксис

```
if logic_expression:  
    do_some_work
```

ATTENTION! Логика, которая будет выполняться в случае истинности выражения определяется в новом программном блоке!

```
if True:  
    # do something  
    print('Do a lot of work')
```

Условные операторы if ... else ...

Синтаксис

if logic_expression:

do_some_work

else: # если выражение не верно

do_other_work

```
19 i = 1
20 if i > 10:
21     print("i is greater than 10")
22 else:
23     print("i is less than 10")
24
25
```

Условные операторы (switch case - like)

Синтаксис

if logic_expression:

 do_some_work

elif other_logic_expression:

 do_other_work

else:

 do_smth_else

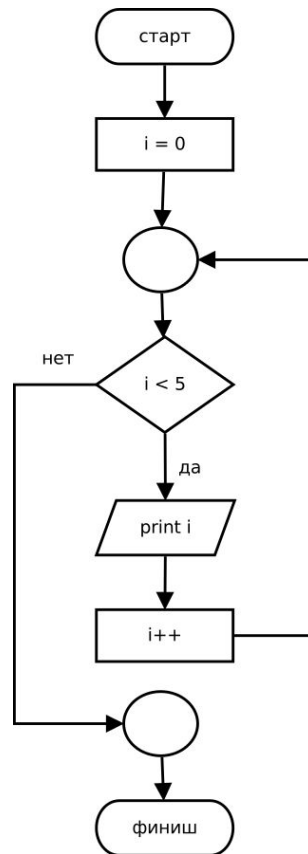
```
26 i = 20
27 if i > 30:
28     print("i is greater than 30")
29 elif i < 30 and i >= 20:
30     print("i is less than 30 but it greater or equal to 20")
31 else:
32     print("To small value for i")
```

DRY принцип

DRY - don't repeat yourself. Принцип говорит нам о том, что если вы заметили в вашем коде места, которые принципиально делают одну и ту же работу но различаются лишь данными над которыми они работают, то нужно такие места локализовывать и выносить в функции или циклы (в зависимости от ситуации).

Циклы

Цикл — разновидность управляющей конструкции в высокоуровневых языках программирования, предназначенная для организации многократного исполнения набора инструкций.



Как реализовать цикл в Python

Python реализует два способа создания цикла:

- **while** - предполагает повторное выполнение кода внутри цикла, до тех пор пока выполняется некоторое условие
- **for** - предполагает повторное выполнение кода внутри цикла, до тех пор пока не будут перебраны все элементы некоторой последовательности (список, словарь, генератор, range объект...)

while

Синтаксис

while <logic_expression>:
 do_smth

ATTENTION! Логика определяющая работу цикла определяется в новом программном блоке - еще называется **телом цикла**

```
1  i = 1
2  while i < 10:
3      print(i)
4      i += 1
5
```

for

Синтаксис

for <var> **in** <collection>:
 do_some_work

```
7  l = [1,2,3,4,5,6,7,8,9]
8  for i in l:
9      print(l)
10
11 for i in range(1, 10, 1):
12     print(l)
```

while/for ... else

Если дополнительно нужно обработать ситуацию запланированного завершения цикла, то можно использовать конструкцию.

```
In [18]: i = 1  
  
In [19]: while i < 10:  
...:     print(i)  
...:     i += 1  
...: else:  
...:     print("End")  
...:
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
End
```

```
In [20]: for i in range(5):  
...:     print(i)  
...: else:  
...:     print('Finish')  
...:  
0  
1  
2  
3  
4  
Finish
```

Ключевое слово **break**

Если нужно принудительно прекратить итерацию по циклу, то можно использовать ключевое слово **break**

```
1 i = 1
2 while i < 10:
3     if i % 5 == 0:
4         print(i)
5         break
6     i += 1
```

```
In [16]: i = 1

In [17]: while i < 10:
...:     print(i)
...:     i += 1
...:     if i == 5:
...:         break
...: else:
...:     print("End")
...:
```

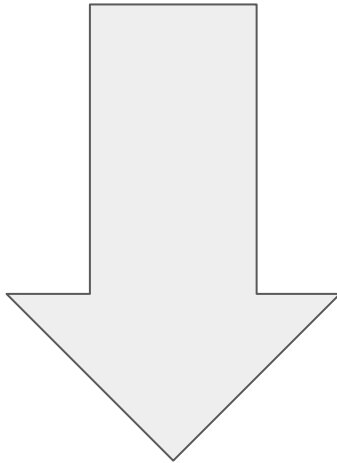
```
1
2
3
4
```

Ключевое слово **continue**

В ситуациях, когда вы предполагаете, что в каком-то месте тела цикла нужно прервать итерацию и начать следующую нужно использовать ключевое слово **continue**. Работает и для `for` и для `while`

```
In [21]: for i in range(10):  
...:     if i % 2 == 0:  
...:         continue  
...:     print(f"{i} is odd")  
...:  
1 is odd  
3 is odd  
5 is odd  
7 is odd  
9 is odd
```

Практическая Часть



Циклы

1. Выведите последовательность чисел 1..10 двумя различными типами цикла (while и for)
2. Выведите последовательность квадратов чисел от 1..100 различными типами цикла (while и for)
3. Выведите последовательность чисел от 112..133 в обратном порядке различными типами цикла (while и for)
4. Выведите последовательность Фибоначчи до 14ого числа
5. Выведете геометрическую последовательность до 15 числа (начиная с 1, шаг определяйте как хотите)
6. Посчитайте сумму предыдущей последовательности
7. Выведете матрицу некоторых числовых значений (на ваш выбор), двумя типами циклов (while, for)
8. Выведете некоторую диагональную матрицу
9. Выведете некоторую треугольную матрицу

Условные операторы

1. Напишите условие проверяющее, что 200 больше 100 и меньше 300 двумя способами (с помощью `and` и `or` и как цепочку выражений)
2. Напишите условие проверяющей, что введено число больше 10 (число вводится при помощи команды `input`)
3. Напишите проверку на четность числа
4. Что будет результатом вычисления этого выражения $(0 \text{ if } x < 0 \text{ else } 1) + 1$, объясните почему
5. Коля купил `N` пирожков, а у Вани 100 пирожков напишите условие, проверяющее, что общее количество пирожков у ребят больше 100 и не больше 150 или больше 200 но меньше 300. Количество пирожков для Коли вводится из командной строки (`input`).
6. Предположим, что вы ввели число от 1..5 тогда напишите ряд условий проверяющий что это за число и выводящий его слово на английском или русском обозначающий это слово (например 1-one или один)
7. Выведете таблицы истинности для выражений `x and y` `x or y` `x and not y` `(x or z) and (y or z)`
8. Как может выглядеть вечный цикл

Циклы + Условные операторы

1. Напишите проверку числа на простоту (число является простым, если оно делится нацело только на себя и единицу)
2. Выведите все простые числа для заданного интервала
3. Выведите все числа в заданном интервале, позиция которых четна/нечетна (режим работы должен определяться из некоторой переменной которая при значении True должны выводить числа стоящие на четной позиции и наоборот).
4. Запустите вечный цикл при этом на каждом шаге цикла просите ввести некоторое значение пользователя (команда `input(...)`), если пользователь ввел букву `q` то ваша программа должна завершаться
5. Найдите сумму всех четных элементов ряда Фибоначчи, которые не превышают четыре миллиона.
6. Найдите сумму всех чисел меньше 1000, кратных 3 или 5.
7. Найдите все тройки пифагора для заданного интервала

Практическая Часть

К этому заданию нужно приступить после выполнения заданий выше
Реализовать программу чат бот. Программа должна уметь общаться с пользователем
реагировать на набор заданных фраз:

```
[19:18:57] serg :: serg-pc → ~/tmp»  
python3 chat_bot.py  
Put you'r question here: привет  
Hello  
Put you'r question here: Как дела?  
Super! Thanks for asking  
Put you'r question here: ты кто?  
I don't understand. Please rephrase your message.  
Put you'r question here: пока  
Bye  
[19:19:25] serg :: serg-pc → ~/tmp»
```