Python

ЯЗЫК ПРОГРАММИРОВАНИЯ
СВЕРХВЫСОКОГО УРОВНЯ
ОБЩЕГО НАЗНАЧЕНИЯ

Содержание

Языки программирования

С КАКОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ НАЧАТЬ ОБУЧЕНИЕ?

ЧТО ТАКОЕ ПРОГРАММИРОВАНИЕ?

Написание крайне специфичных инструкций глупой, но очень послушной машине

ЯЗЫКИ

PYTHON · JAVA · C · PHP · C++
JAVASCRIPT · C# · RUBY · OBJECTIVE-C

СТАРТ

Зачем вы хотите изучить программирование?

- Для детей → Начните со Scratch, затем...
- Я не знаю, выберите за меня
- Поразвлечься
- Интересно
- Саморазвитие
- Заработать → Найти работу
- У меня есть идея для стартапа!

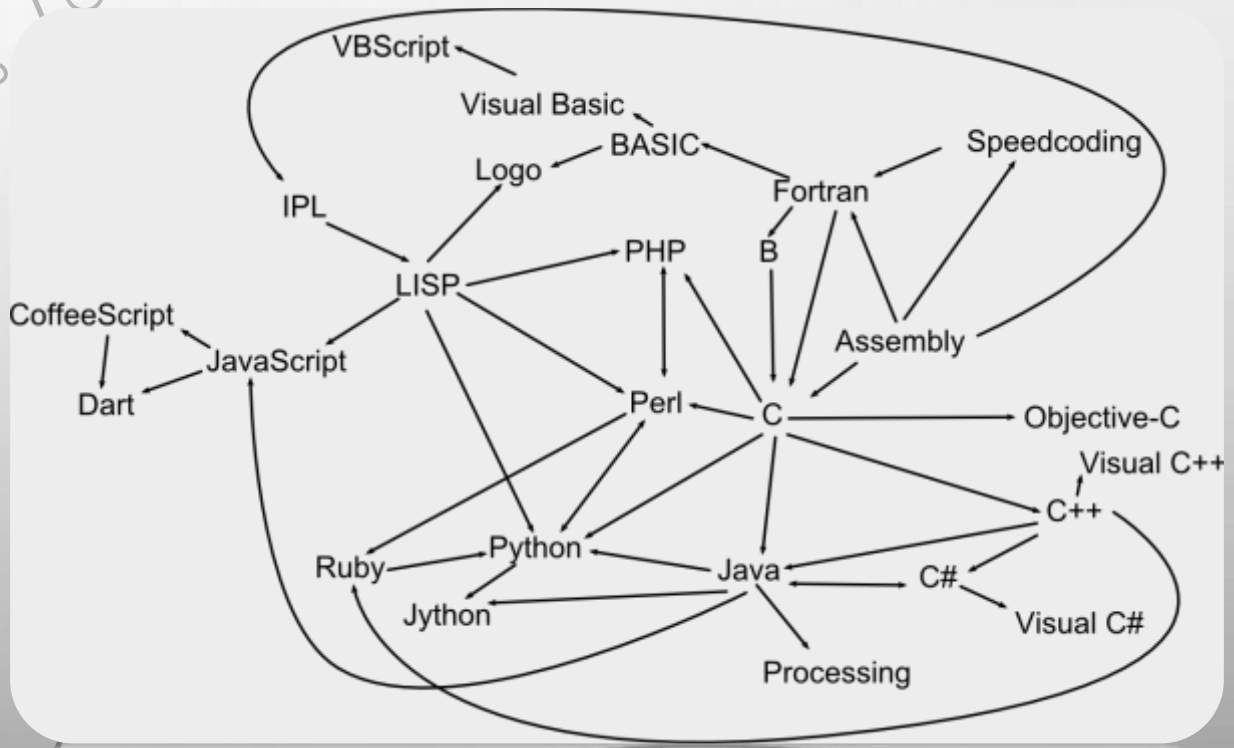
Уже есть идея на миллион?

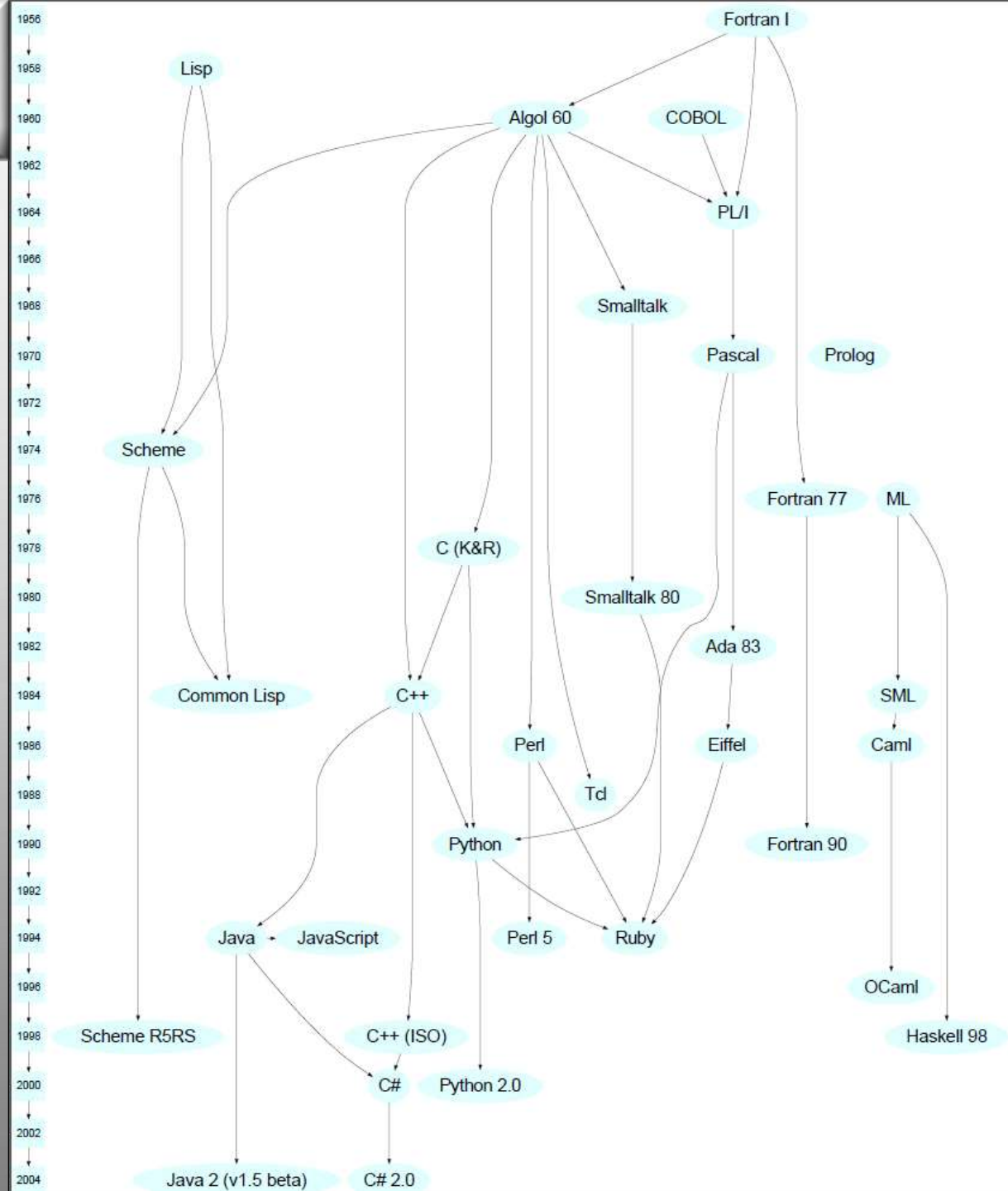
- Да → Какая платформа/сфера?
- Нет. Просто хочу начать → Мне нравится учиться
  - Простым способом
  - Лучшим способом
  - Трудненьким способом → Какая коробка передач?
    - Автомат
    - Ручная

Какая платформа/сфера?

- Я хочу работать в крупной IT-компании
- Я просто хочу $$$
- Веб → Бэкенд (то, что за вебсайтом) → Хочу работать на...
  - Корпорацию → Что скажете о Microsoft?
    - Люблю его
    - Неплохо
    - Фууу
  - Стартап
- Фронтенд (веб-интерфейс)
- 3D/игры
- Мобильная → Какая OS?
  - iOS
  - Android
- Корп.софт
- Веб → Ваш сервис будет работать в реальном времени, как twitter?
  - Да
  - Нет
- Хотите попробовать что-то новое, но не очень трудоёмкое?
  - Не знаю
  - Нет → Какая у вас любимая игрушка?
    - Lego
    - Пластилин
    - Старенький, но любимый мишка
- Очень сложный путь (но потом будет легче)

Эволюция языков программирования





Поколения языков программирования

1GL	набор машинных команд в двоичном или восьмеричном формате, который определялся архитектурой конкретной ЭВМ	HEX, binary
2GL	языки ассемблерного типа, позволяющих вместо двоичных и других форматов машинных команд использовать их мнемонические символьные обозначения (имена)	Assembler
3GL	их основные характеристики: относительная простота, независимость от конкретного компьютера и возможность использования структурных синтаксических конструкций ориентированных на алгоритм.	Fortran, C, Pascal
4GL	ориентированы на специализированные области применения, не универсальны, а проблемно-ориентированы, описывают что сделать, а не как.	SQL, HTML, JavaScript
5GL	Объектно-ориентированные универсальные системы автоматического создания прикладных программ с помощью визуальных средств разработки.	C#, Python, Java

Исполнение программ

Язык высокого уровня

```
# Python 3: Fibonacci series up to n
>>> def fib(n):
>>>     a, b = 0, 1
>>>     while a < n:
>>>         print(a, end=' ')
>>>         a, b = b, a+b
>>>     print()
>>> fib(1000)
```

Язык низкого уровня

```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text           segment para public 'CODE' use32
                assume cs:_text
                ;org 100101Ch
                assume es:nothing, ss:nothing, ds:_text, fs:nothing, gs:nothing
                dd 2 dup(0)
                dd 3B7D845Bh, 0
                dd 2, 10h, 1048h, 448h
; ---------------------------------------------------------------------------

; const CHAR szFile
_szFile:                                        ; DATA XREF: WinMain(x,x,x,x)+Bjo
                jb      short near ptr loc_100102A2+1
                db      65h
                imul    esi, fs:[si+2Eh], 657865h
                dec     esi
                inc     edx
                xor     [eax], esi
; ---------------------------------------------------------------------------
                dd 0
                dd 3B7D845Bh, 1, 65676572h, 32337464h, 6264702Eh
                db 2 dup(0)
```

Машинный код

```
00000270: 84 D3 2B 38-86 00 00 00-
00000280: 46 22 D1 38-9B 00 00 00-0
00000290: 63 6F 6D 64-6C 67 33 32-2
000002A0: 4C 4C 33 32-2E 64 6C 6C-0
000002B0: 64 6C 6C 00-41 44 56 41-5
000002C0: 00 4B 45 52-4E 45 4C 33-3
000002D0: 49 33 32 2E-64 6C 6C 00-5
000002E0: 6C 6C 00 57-49 4E 53 50-4
000002F0: 00 00 00 00-00 00 00 00-0
00000300: EB 02 EB 05-E8 F9 FF FF-F
00000310: FC FF FF 83-E4 FC 8B EC-3
00000320: 00 40 E2 FA-E8 60 00 00-0
00000330: 41 64 64 72-65 73 73 00-
```

Выполнение кода

Исполнение программ

Транслятор

Преобразовывает программы, представленные на одном из языков программирования, в программы на другом языке и, в определённом смысле, равносильную первой.

Компилятор

Транслирует программу, составленную на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду

Интерпретатор

Анализирует и тут же выполняет программу покомандно, по мере поступления её исходного кода на вход интерпретатора

Парадигмы программирования

Императивное

описывает процесс вычисления в виде инструкций, изменяющих состояние данных. Это последовательность команд, которые должен выполнить компьютер.

Процедурное

Fortran, COBOL

Является отражением архитектуры традиционных ЭВМ Фон Неймана и Алана Тьюринга. Выполнение сводится к последовательному выполнению операторов, преобразующих исходное состояние памяти в результат. Блоки кода можно собрать в подпрограммы – более крупные логически целостные единицы.

Структурное

Java, Pascal

Программа строится без использования оператора goto из трёх базовых управляющих структур: последовательность, ветвление, цикл. При этом разработка программы ведётся пошагово, в виде иерархической структуры блоков, методом «сверху вниз», используя подпрограммы для структурирования.

Объектно-ориентированное

C++, Python, Java, C#

Программа представлена в виде совокупности объектов (а не алгоритмов!), каждый из которых является экземпляром определенного класса, при этом классы образуют иерархию наследования.

Парадигмы программирования

Декларативное

SQL, HTML

описывает не порядок решение задачи, а результат, который нужно получить, при этом метод решения определяет компьютер. Не содержит понятие «состояние», переменных и присваивания.

Функциональное

Scala, F#, Haskell

процесс вычисления трактуется как вычисление значений функций в математическом понимании, т.е. предполагает обходиться вычислением результатов функций от исходных данных, и не предполагает явного хранения состояния программы.

Логическое

Prolog

Основано на языке предикатов математической логики применяя правила вывода на основе заданных фактов. Программы выражаются в терминах отношений, представленных в виде фактов и правил. Для того, чтобы инициировать вычисления, выполняется специальный запрос к базе знаний, на которые система логического программирования генерирует ответы «истина» и «ложь»

Свойства и преимущества

- **М**инимализм и простота кода → *стимулирует писать хорошо читаемый код*.
- **Н**аличие интерактивного режима → *позволяет на лету тестировать нужные участки кода*.
- **П**опулярный и активно используемый → *4-е место в рейтинге TIOBE, применятся в Google, Яндекс, YouTube, NASA, и др…*
- **М**ульти-платформенный → *работает под Windows, Linux, Android, UNIX, MacOS, iOS, …*
- **М**ульти задачный → *разработка GUI, web, администрирование ОС, мобильные приложения, научные расчёты и анализ данных, разработка игр, и др…*
- **М**ультипарадигменный → *ООП, структурное, функциональное, …*
- **А**ктивно развивается → *версия 3.6.0 выпущена 16 мая 2016 года*
- **И**нтерпретируемый язык с несколькими реализациями → *CPython (эталон), PyPy, Jython, IronPython, и др…*
- **Б**огатая встроенная библиотека и огромное количество внешних.
- **А**бсолютно всё в **Python** является объектами, но при этом объектный подход не навязывается
- **Х**орошо документирован (in English), много обучающих материалов.

Особенности языка

Разработка для нескольких платформ:

- Кроссплатформенные GUI-приложения для Windows, либо Linux (модуль **tkinter**).
- Серверные Web-приложения (framework **Django**).
- Приложения для мобильных устройств Android и iOS (модуль **kivy**).
- Администрирование операционных систем Windows, либо Linux (модуль **os**).
- Анализ “больших” данных, научные вычисления и научная графика (модули **numpy**, **scipy**, **matplotlib**).
- Разработка игр на основе OpenGL (модуль **pygame**).

Архитектурные черты:

- Динамическая типизация,
- Автоматическое управление памятью,
- Полная интроспекция,
- Механизм обработки исключений,
- Поддержка многопоточных вычислений
- Удобные высокоуровневые структуры данных

Преимущества для изучения

- Используется многими учебными заведениями США и Европы, как первый изучаемый язык программирования.
- С 2015 года является одним из стандартных языков программирования в **ЕГЭ по информатике**.
- Применяется на **олимпиадах** по информатике всех уровней.
- Существует широкий выбор **материалов и курсов** по изучению языка:

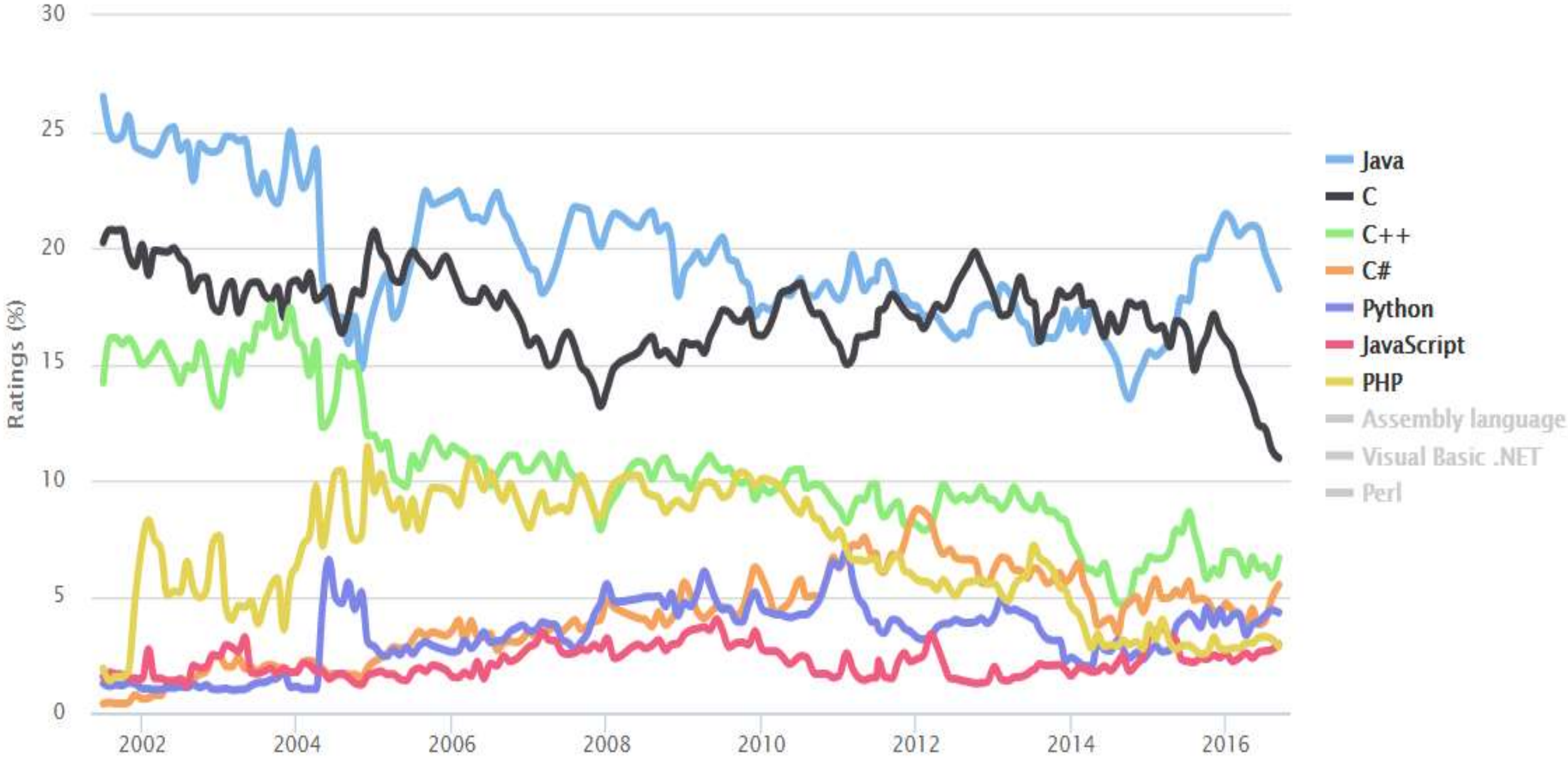
  pythontutor.ru – курс, включающий теорию, много практических задач для закрепления материала и удобную систему отладки.

  checkio.org, codewars.com, codingame.com, – онлайн-игры, с решением пазлов и обменом опыта.

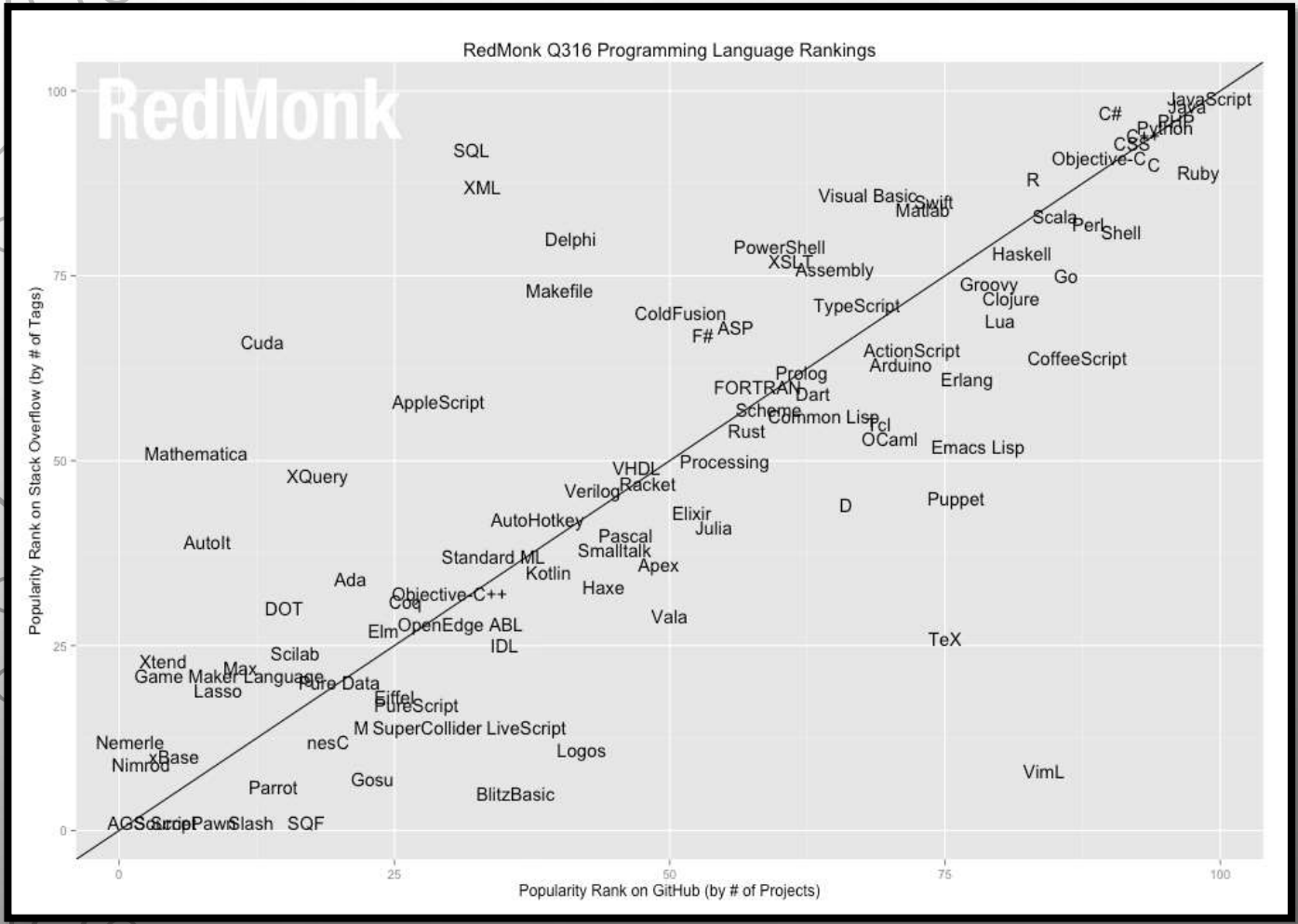
  stepik.org#1, stepik.org#2 – курсы для новичков, с заданиями и видео-уроками.

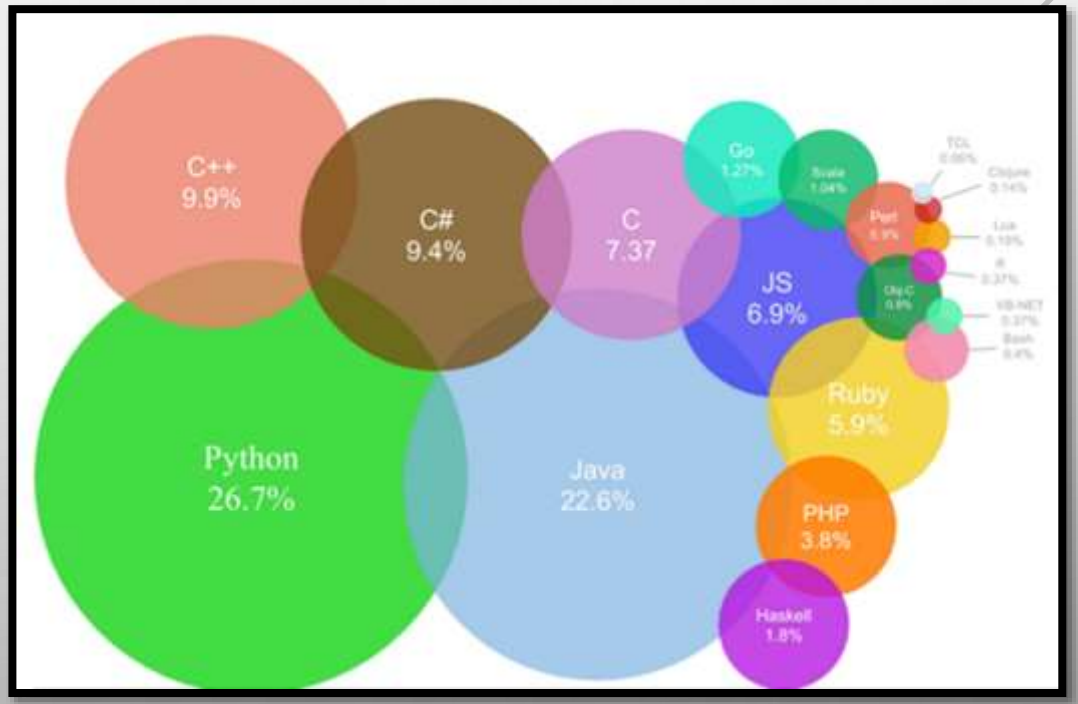
  coursera.org – курс «Python для анализа данных».
- **Книга:** Марк Саммерфилд «Программирование на Python 3»

Популярность



Популярность





1. Tiobe.com
2. RedMonk.com
3. Spectrum.IEEE.org
4. Github.io
5. Githut.info

История создания

python → python™

1989 → 1994 → 2000 → 2008

Гвидо Ван Россум

- **1989.** Начало работы над созданием Python, в основу которого лёг язык ABC, предназначенный для обучения программированию, работа шла в центре математики и информатики (Нидерланды).
- **1994.** Выпуск версии 1.0, включившей методы функционального программирования.
- **2000.** Выпуск версии 2.0, добавлены генераторы, обновлён сборщик мусора, полный переход на объектно-ориентированную модель.
- **2008.** Выпуск версии 3.0, устранение фундаментальных изъянов – дубликатов, укрепление мультипарадигменности.

MONTY PYTHON'S FLYING CIRCUS

- Гвидо Ван Россум является основным автором Python и по сей день продолжает выполнять центральную роль в принятии решений относительно развития языка.
- Язык назван в честь популярного британского комедийного телешоу 1970-х «**Летающий цирк Монти Пайтона**».

Дзен Питона

- Красивое лучше, чем уродливое.
- Явное лучше, чем неявное.
- Простое лучше, чем сложное.
- Сложное лучше, чем запутанное.
- Плоское лучше, чем вложенное.
- Разреженное лучше, чем плотное.
- Читаемость имеет значение.
- Должен существовать один — и, желательно, только один — очевидный способ сделать это.
- Если реализацию сложно объяснить — идея плоха.
- Если реализацию легко объяснить — идея, возможно, хороша.
- Пространства имён — отличная штука! Будем делать их побольше!
- Особые случаи не настолько особые, чтобы нарушать правила.
- При этом практичность важнее безупречности.
- Ошибки никогда не должны замалчиваться.
- Встретив двусмысленность, отбрось искушение угадать.

18 Среда разработки, IDE

Python IDLE

Python 3.5.1 Shell

File Edit Shell Debug Options Window Help

```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25) [MSC v.1900 64 bit (
AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
========================== RESTART: F:\1\fib.py ==========================
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>>
```

fib.py - F:\1\fib.py (3.5.1)

File Edit Format Run Options Window Help

```
def fib(n):
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b

fib(1000)
```

http://repl.it/languages/python3

repl.it - Python3 Compile

https://repl.it/languages/python3

repl.it Untitled Session Log in

```
1 def fib(n):
2     a, b = 0, 1
3     while a < n:
4         print(a, end=' ')
5         a, b = b, a+b
6
7 fib(1000)
```

Input clear

```
Python 3.5.1 (default, Dec 2015, 13:05:11)
[GCC 4.8.2] on linux
Listing 'F:/1'...
Can't list 'F:/1'
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
=> None
```

Алфавит языка

Прописные и строчные буквы	A B C … A Б B … a b c … a б в …
Арабские цифры	0 1 2 3 4 5 6 7 8 9
Арифметические операции	+ - * / // %
Знаки присваивания	= += -= *= /= //= %=
Специальные символы	( ) [ ] { } ' ' " " ; :
Знаки сравнения	> < == != <= >=
Комментарий	# comment '''comment'''

ПРОПИСНЫЕ и строчные буквы различны!!!

ASCII

&

Unicode

Словарь языка

Служебное слово	Значение
if <условие>:	Если
elif <условие>:	Иначе если
else:	Иначе
while <условие>:	Пока выполняется условие
for <переменная> in <итератор>:	Для элемента во множестве
break	Прервать
continue	Продолжить
True	Правда
False	Ложь
None	Пустое значение

Служебное слово	Значение
<объект> and <объект>	Логическое И
<объект> or <объект>	Логическое ИЛИ
not <объект>	Логическое НЕ
<объект> is <объект>	Проверка на соответствие
def <имя>(<параметры>):	Объявление функции
class <имя>:	Объявление класса
return <объект>	Вернуть
<объект> as <имя>	(что-то) как (имя)
del <объект>	Удалить
pass	Пропуск действия

Имена переменных

Имена (констант, переменных, функций, классов):

- Могут состоять только из букв, цифр и знака нижнего подчёркивания «_»
- Не должны совпадать со служебными словами
- Не могут начинаться с цифры
- Длина имени не может превышать 255 символов.

Желательно давать осмысленные имена!!!

Правильные имена (PEP8):

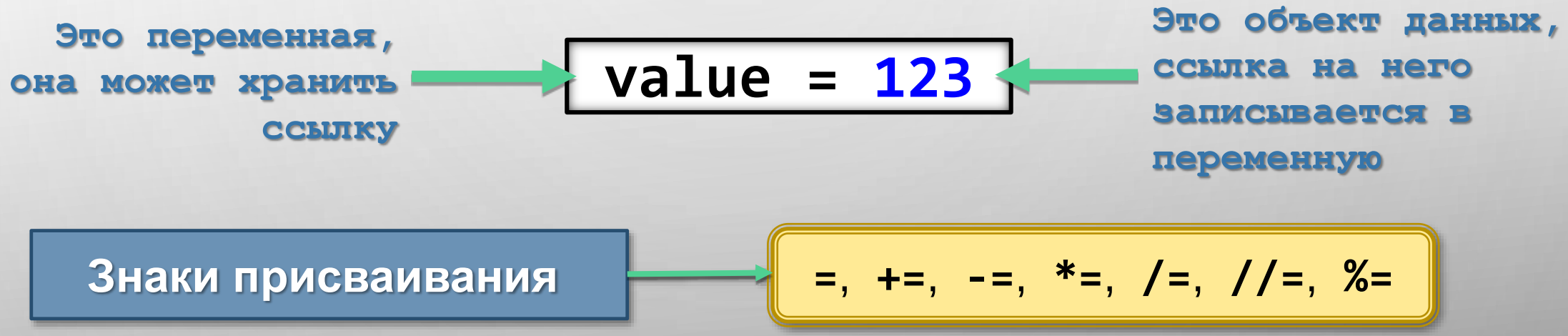
```
value                  # имя переменной
kolichstvo_tochek      # ещё имя переменной
poisk_max              # имя функции
MyClass                # имя класса
LENGTH                 # имя константы
# и не очень…
x, s25                 # не ясен смысл
a1b88qq                # смысла нет!
```

Неправильные имена

```
# содержится пробел
polnaja summa
# начинается с цифры
2parent
# содержит спец.символы
Domby&Son
```

Операция присваивания

Операция помещает объект данных (справа от знака =) в именованную переменную (слева от знака =).




```
country = "Russia“     # переменной country присвоено значение типа str
age = 23               # переменной age присвоено значение типа int
phone = '89211234567‘  # переменной phone присвоено значение типа str
weight = 83.2          # переменной weigth присвоено значение типа float
growth = 187.          # переменной growth присвоено значение типа float
is_male = True         # переменной is_male присвоено значение типа bool
```

Операция присваивания

Присваивание с выполнением арифметических операций:

сложение `+=`, вычитание, `-=`, умножение `*=`, деление `/=`, целочисленное деление `//=`, остаток от деления `%=`

```
value = 1          1
value += 10        11
value -= 2         9
value //= 2        4
value *= 4         16
value %= 10        6
value /= 2         3.0
```

Множественное присваивание значений:

Многократное присваивание

Присваивание с помощью кортежа

```
a = b = c = 2
print(a + b * c)      6

d, e = 2, 3
print(d + e)          5
```

Обмен значений между переменными:

Вариант 1

```
a = 2
b = 3
c = a
a = b
b = c
```

Вариант 2

```
a = 2
b = 3
a, b = b, a
```

Простые типы данных

Название	Пример	Операции	Результат	Преобразование типов

Целые числа
int
	`n = 3`
	`m = -63`
		`>>> 3 + (-63)`
		`>>> int(45.6)`
		`>>> int('бдыщ')`
			`-60`
			`45`
			`Error`
				int()
				Преобразует переданную ей строку или число с плавающей точкой в целое

Вещественные числа
float
	`f = 1.34`
	`d = -3.4352`
	`s = 3.`
		`>>> 34.907 + 320.65`
		`>>> 1 + 0.65`
		`>>> float(34)`
			`355.55699999996`
			`1.65`
			`34.0`
				float()
				Преобразует переданную ей строку или целое в число с плавающей точкой

Строки
str
	`str = 'Hello'`
	`str_num = '45'`
	`pass = "kdKEJs47"`
		`>>> 'Hi, ' + 'world!'`
		`>>> 'Hi ' * 4`
		`>>> '3 ' + 15`
		`>>> str(45.4)`
			`'Hi, world! '`
			`'Hi Hi Hi Hi '`
			`Error`
			`'45.4'`
				str()
				Преобразует переданный ей аргумент в строку

Логические
bool
	`is_male = True`
	`ok = False`
		`>>> True or False`
		`>>> False and True`
		`>>> 2 > 1`
			`True`
			`False`
			`True`
				bool()
				Преобразует переданный ей аргумент в логическое значение

Арифметические операции

a + b	Сложить **a** и **b**
a - b	Вычесть **b** из **a**
a * b	Умножение **a** на **b**
a / b	Поделить **a** на **b**
a // b	Целочисленно поделить **a** на **b**
a % b	Получить остаток от деления **a** на **b**
a ** b	Возведение числа **a** в степень **b**

```
>>> 2 + 2 * 2
6
>>> (17 - 2) * 8
120
>>> 2 ** 8
256
>>> 10 / 3
3.333334
>>> 20 // 3
6
>>> 20 % 3
2
>>> 150 + 17.3
167.3
>>> 'ab' * 3
'ababab'
>>> 'ab' + 'cd'
'abcd'
```

битовые операции

`x | y`	Поразрядное логическое сложение. Логическое ИЛИ
`x & y`	Поразрядное логическое умножение. Логическое И
`x ^ y`	Поразрядное сложение по модулю 2. Логическое исключающее ИЛИ
`~x`	Поразрядная инверсия. Логическое НЕ
`x >> y`	Сдвиг значения X вправо на количество бит, равное Y
`x << y`	Сдвиг значения X влево на количество бит, равное Y

```
>>> 0b10010 | 0b111
23    0b10111
>>> 0b1110 & 0b101
4     0b100
>>> 0b1110 ^ 0b101
11    0b1011
>>> ~0b1010
-11   -0b1011
>>> 0b1010 >> 1
5     0b101
>>> 0b101 << 2
20 0b10100
```

Вывод данных

Вывод данных из оперативной памяти на экран монитора осуществляется с помощью функции:

```
print([значение[, значение]*][, sep=строка][, end=строка][, file=объект_файла])
```

Дополнительные опции (*записываются после значений на вывод*):

sep – это символьная строка, размещаемая между значениями (по умолчанию пробел ' ').

end – это символьная строка, размещаемая в конце выводимого на печать текста (по умолчанию символ перевода строки '\n').

file – это файловый объект, если опция указана, то данные записываются в него вместо вывода на экран

```
print('Hello, World!')
print(2017)
print('Hi', 'Jack')
print(123, 0.123)
print(3 * 7, (17 - 2) * 8)
print('Hi! ' * 5)
```

```
'Hello, World!'
2017
'Hi Jack'
123 0.123
21 120
'Hi! Hi! Hi! Hi! Hi! '
```

Вывод данных

```
print([значение[, значение]*][, sep=строка][, end=строка][, file=объект_файла])
```

Варианты вывода	Оператор вывода	Результат

Без разделителей	`print('Hello' + 'World!')`
	`print("Hello", "World!", sep='')`
	`print(1, 2, 3, sep='')`
		`HelloWorld!`
		`HelloWorld!`
		`123`

Разделители – запятые	`print(1, 2, 3, sep=',')`	`1,2,3`
Разделители – пробелы	`print(1, 2, 3)` `print("Hello", "World!")`	`1 2 3` `Hello World`
Однострочный вывод	`print('2 + 2 * 2 = ', end='')` `print(2 + 2 * 2)`	`2 + 2 * 2 = 6`

Функции форматирования строк

Функции	Код	Результат

str.format(v1[, v2]*)
форматирование строки по шаблону **str**, вставляя значения в скобках **v1**, **v2** и т.д.... в специальные места, обозначенные фигурными скобками
	`print('{} + {} = {}'.format(1,2,3))`
	`print('{0:.3f}'.format(math.pi))`
	`print('{0:10d}'.format(123))`
	`print('{0:10} + {1}'.format('абвгд','ёпрст'))`
		`1 + 2 = 3`
		`3.142`
		`       123`
		`абвгд      + ёпрст`

str.zfill(n) Заполнение строки **str** с левой стороны, нулями до длины строки в **n** символов	`print(str(0).zfill(2), ':', \` `  str(7).zfill(2), sep='')`	`00:07`

str.rjust(n)
str.ljust(n)
str.center(n)
выравнивание строки **str** по одному из краёв (правому, левому, либо по центру) в поле ширины **n**, заполняя пустоты пробелами
	`print('[', str(2**3).center(5), '][', \`
	`  str(2**4).ljust(5), '][', \`
	`  str(2**5).rjust(5), ']', sep='')`
		`[  8  ][16   ][   32]`

30 Вывод данных в файл

```
print([значение[, значение]*][, sep=строка][, end=строка][, file=объект_файла])
```

Создание файловой переменной (т.е. связывание файла с переменной):

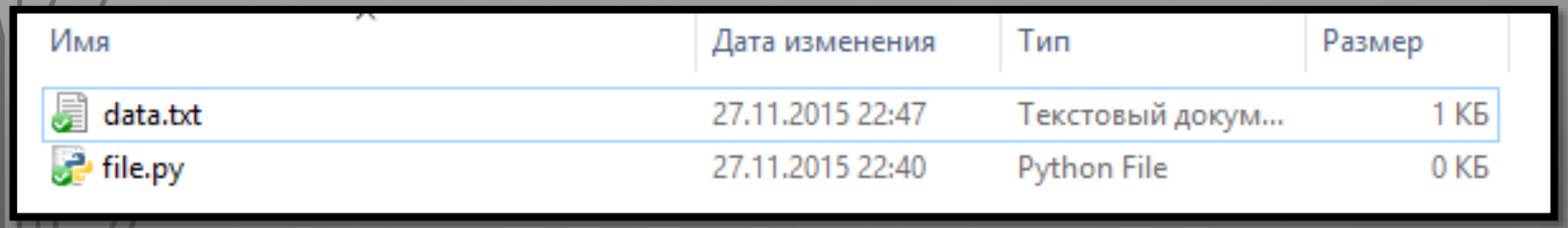
```
file_obj = open(имя_файла [, режим_открытия])
```

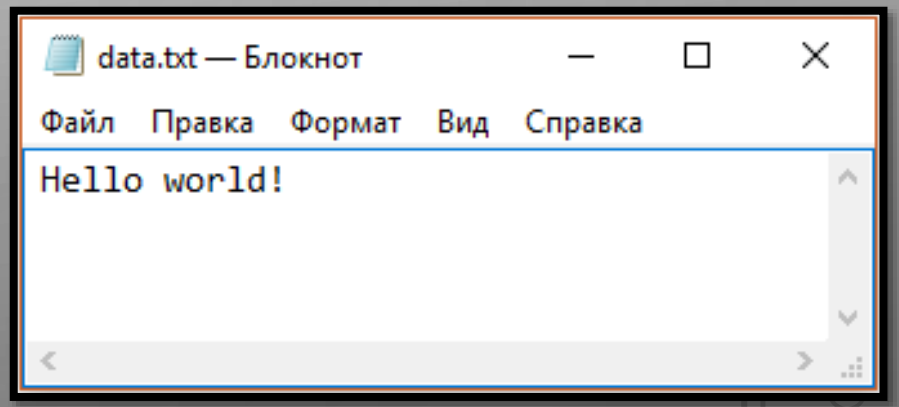
Режимы открытия:

1. `'r'` (по умолчанию) – открытие на чтение
2. `'w'` – открытие на запись
3. `'a'` – открытие на добавление

file.py

```
file_obj = open('data.txt', 'w')
print('Hello, World!', file=file_obj)
file_obj.close()
```





Ввод данных

Для ввода данных с клавиатуры служит функция `input()`,
она считывает введённые данные, возвращая их в виде строки

```
data_in = input([строка приглашения к вводу])
```

```
print('Введите ваше имя:', end=' ')
name = input()
print('Здравствуйте,', name)
```

```
name = input('Введите ваше имя: ')
print('Здравствуйте,', name)
```

```
Введите ваше имя: Евгения Александрович
Здравствуйте, Евгений Александрович
```

Вариант 1

```
a = input()
b = input()
s = a + b
print(s)
```

```
1
2

12
```

Вариант 2

```
a = int(input())
b = int(input())
s = a + b
print(s)
```

```
1
2

3
```

Вариант 3

```
a, b = map(int, input().split())
s = a + b
print(s)
```

```
1 2

3
```

Ввод данных из файла

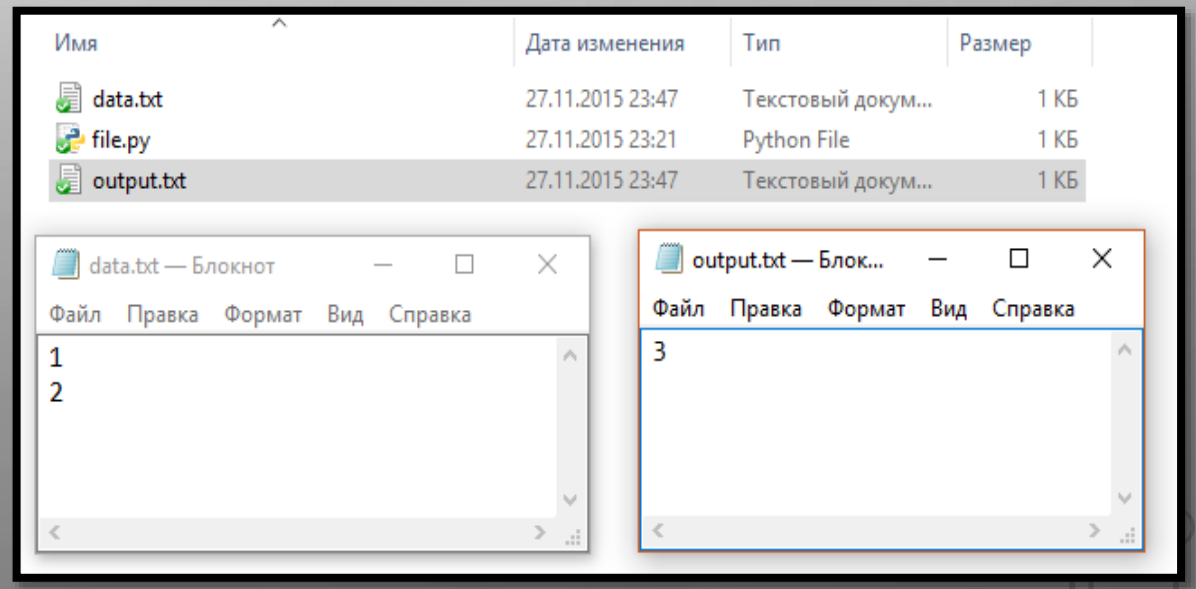
```
file_obj = open('data.txt', 'r')
```

strg = file_obj.read([n]) – чтение файла **file_obj** целиком (либо **n** байт из него) и передача его в переменную **strg** в виде строки.
strg = file_obj.readline() – возвращает текущую строку из файл, перемещая указатель. Чтение может продолжаться вплоть до конца файла.
lst = file_obj.readlines() – возвращает всё содержимое файла в виде списка строк.

Чтение из файла и запись в файл

```
file_in = open('data.txt')
a = int(file_in.readline())
b = int(file_in.readline())
file_in.close()

file_out = open('output.txt', 'w')
print(a + b, file=file_out)
file_out.close()
```



Основы синтаксиса

Отступы и двоеточия

```
from random import choice
if choice([True, False]):
    print('Hi')
else:
    print('Hello')
print('people')
```

```
from random import choice
if choice([True, False]):
    print('Hi')
else:
    print('Hello')
    print('people')
```

```
from random import choice
if choice([True, False]):
    print('Hi')
else:
    print('Hello')
        print('people')
```

Многострочный ввод

```
days = ['Sunday', 'Monday', 'Tuesday',
        'Wednesday', 'Thursday',
        'Friday', 'Saturday']
biography = '''Some long text for few
lines of code'''
print('1.', days[0],
      sep=' ',
      end='.')
total = item1 + item2 + \
    item3 + item4
```

Комментарии и кавычки

```
# first line comment
name = 'Python' # second comment
phrase = 'Hello,'
'''this comment lay down
in few lines'''
print(phrase, name)
```

Разделение команд

```
import random; x = random.randint(0, 10); print(x)
```

Математические функции

Функция	Описание		
Округление			
`int(x)`	Округляет число x отсекая от него дробную часть		
`round(x[, n])`	Округляет число **x** до ближайшего целого (либо до **n** знаков после точки, если **n** указана). Если дробная часть числа равна `0.5`, то число округляется до ближайшего четного числа		
`math.floor(x)`	Округляет число **x** вниз (пол), при этом истинно `floor(1.5) == 1 and floor(-1.5) == -2`		
`math.ceil(x)`	Округляет число **x** вверх (потолок), при этом истинно `ceil(1.5) == 2 and ceil(-1.5) == -1`		
Корни и степени, логарифмы, константы			
`math.sqrt(x)`	Квадратный корень **x**		
`pow(x, y[, z])`	Возвращает **x** в степени **y**, если указан **z**, то результат берётся по модулю **z**		
`math.log(x[, base])`	Возвращает натуральный логарифм числа **x** по основанию **base** (по умолчанию **base** равно **e**, т.е. `2.71`...)		
`abs(x)`	Возвращает модуль числа **x**		
`math.pi`	Константа π = `3.141592653589793`		
`math.e`	Основание натуральных логарифмов **e** = `2.718281828459045`		
Тригонометрия			
`math.sin(x)`	Синус угла **x**, задаваемого в радианах	`math.asin(x)`	Арксинус **x**, возвращает значение в радианах
`math.cos(x)`	Косинус угла **x**, задаваемого в радианах	`math.acos(x)`	Арккосинус **x**, возвращает значение в радианах
`math.tan(x)`	Тангенс угла **x**, задаваемого в радианах	`math.atan(x)`	Арктангенс **x**, возвращает значение в радианах
`math.radians(x)`	Преобразует угол **x**, заданный в градусах, в радианы.	`math.degrees(x)`	Преобразует угол **x**, заданный в радианах, в градусы.

Генератор случайных чисел

Функция	Описание
`from random import *`	
`x_float = random()`	Возвращает случайное вещественное число **f** на диапазоне **[0; 1)**
`x_float = uniform(start, stop)`	Возвращает случайное вещественное число **f** на диапазоне **[start; stop]**
`x_int = randint(start, stop)`	Возвращает случайное целое число **i** на диапазоне **[start; stop]**
`x_int = randrange(start, stop, step)`	Возвращает случайное целое число **i** на диапазоне **[start; stop]** с шагом **step**
`element = choice(sequence)`	Возвращает случайное элемент **element** из последовательности **sequence**
`sequence_res = shuffle(sequence)`	Возвращает последовательность **sequence_res** полученную случайным перемешиванием элементов последовательности **sequence**

```
from random import *
x_float = random()                  # случайное вещественное число на диапазоне [0,1)
x_float = uniform(0, 100)           # случайное вещественное число на диапазоне [0,100]
x_int = randint(0, 100)             # случайное целое число на диапазоне [0,100]
x_int = randrange(0, 100, 2)        # случайное целое число на диапазоне [0,100] с шагом 2
elem = choice([1, 2, 3, 4])         # возвращает случайный элемент из списка
sequence = shuffle([1, 2, 3, 4]) # перемешивает список и возвращает его
```

Генератор случайных чисел

```
from random import *
```

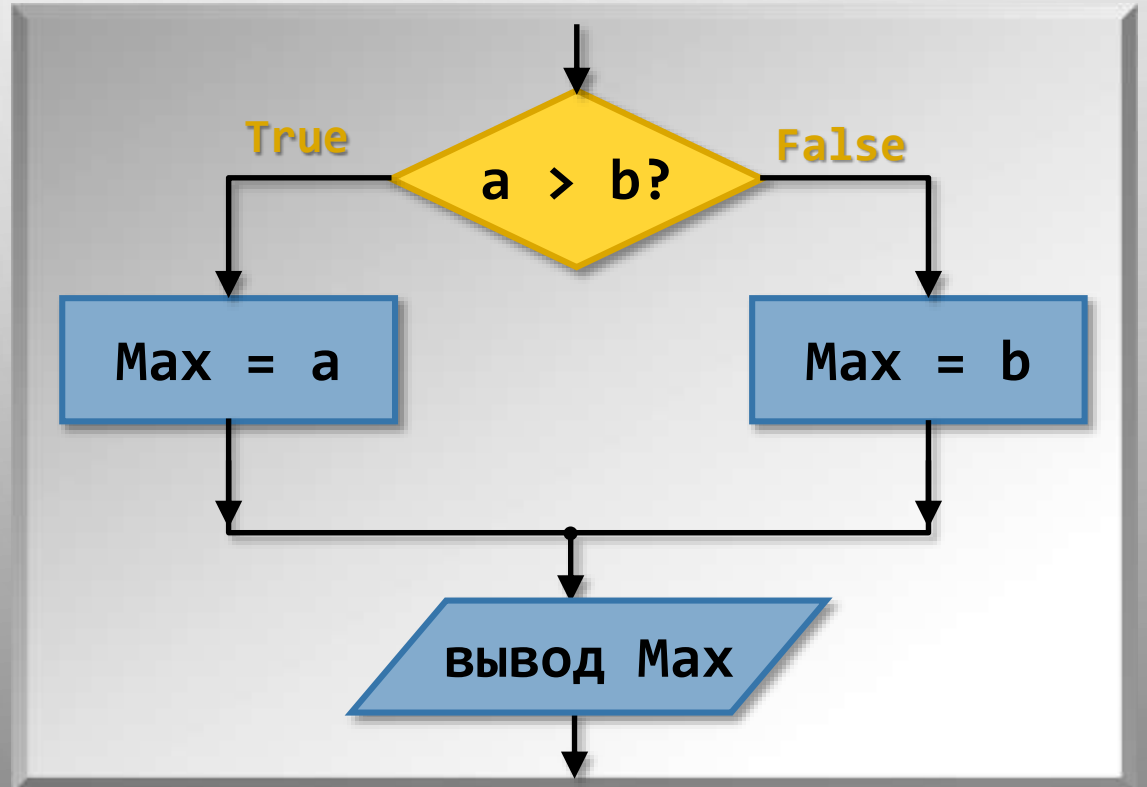
Функция	Описание
Различные статистические распределения (mu – средний угол, sigma – отклонение, alpha – среднее арифметическое, beta -)	
`x = gammavariate(alpha, beta)`	гамма-распределение
`x = gauss(value, sigma)`	распределение Гаусса
`x = lognormvariate(mu, sigma)`	логарифмическое распределение
`x = normalvariate(mu, sigma)`	нормальное распределение
`x = paretovariate(alpha)`	распределение Парето
`x = weibullvariate(alpha, beta)`	распределение Вейбулла

```
from random import *
# различные статистические распределения (mu – средний угол, sigma – отклонение)
x = gammavariate(alpha, beta)   # гамма-распределение.
x = gauss(value, sigma)         # распределение Гаусса
x = lognormvariate(mu, sigma)   # логарифмическое распределение
x = normalvariate(mu, sigma)    # нормальное распределение
x = paretovariate(alpha)        # распределение Парето
x = weibullvariate(alpha, beta) # распределение Вейбулла
```

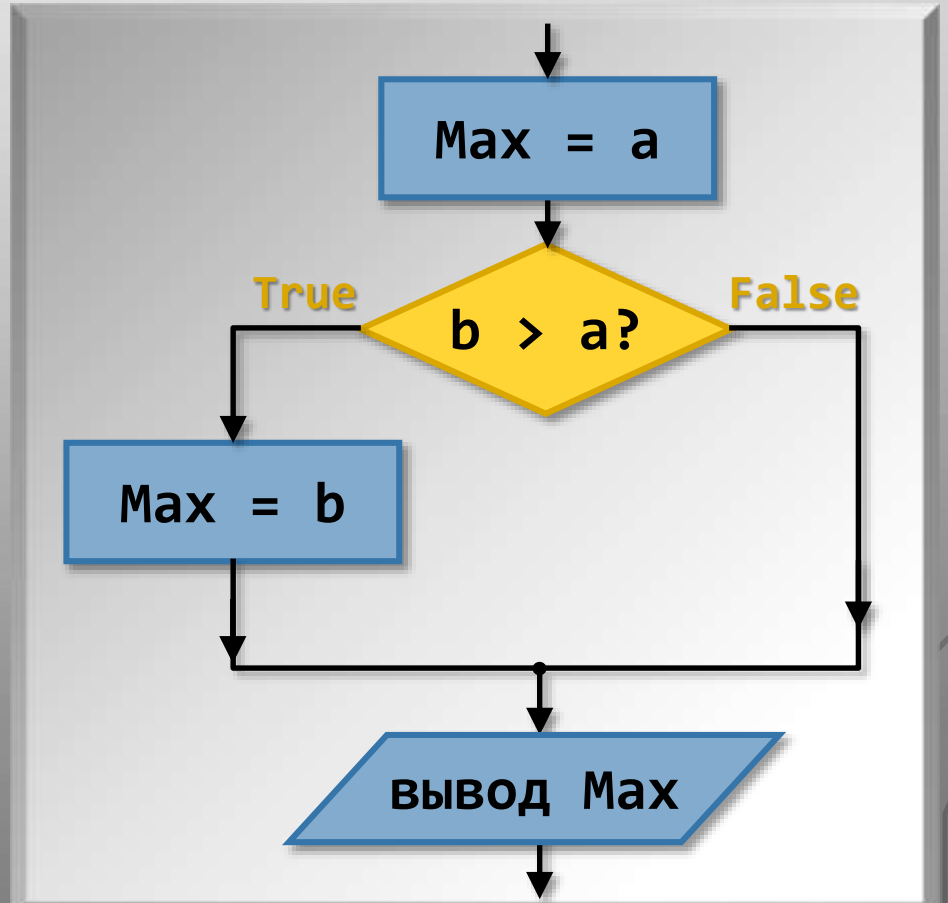
Ветвление. алгоритм

Знаки сравнения → `>, <, ==, !=, <=, >=` `a in b` `a is b`

Полное ветвление



Не полное ветвление



Ветвление. инструкция

```
if <условие>:
    <блок инструкций1>
else:
    <блок инструкций2>
```

Синтаксис условной инструкции.

Если *условие* истинно,

то будет выполнен *блок инструкций1*.

Если *условие* ложно,

то будет выполнен *блок инструкций 2*.

Отступы. Вложенные блоки инструкций выделяются отступом (4 пробели или табуляция **Tab**).

Полное ветвление

```
a = int(input('a = '))
b = int(input('b = '))
if a>b:
    maximum = a
else:
    maximum = b
print(maximum)
```

Не полное ветвление

```
a = int(input('a = '))
b = int(input('b = '))
maximum = a
if b>a:
    maximum = b
print(maximum)
```

Ветвление. Сложные условия

Знаки сравнения

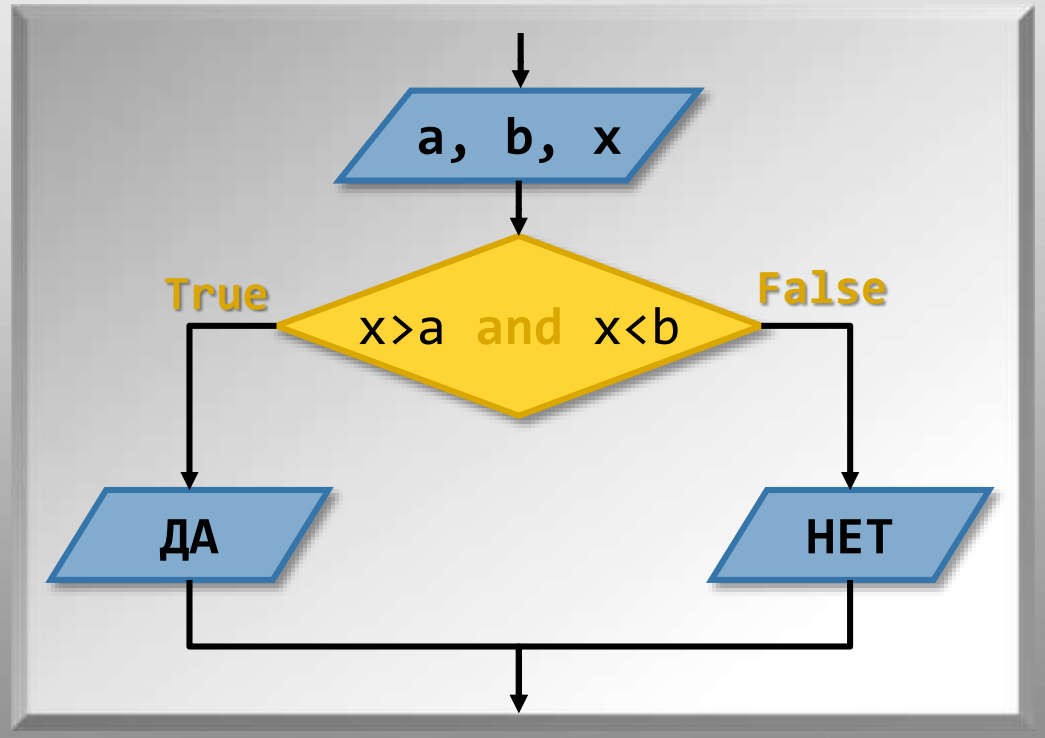
`>, <, ==, !=, <=, >=`

`a in b`

`a is b`

Логические операции

`a and b` `a or b` `not a`

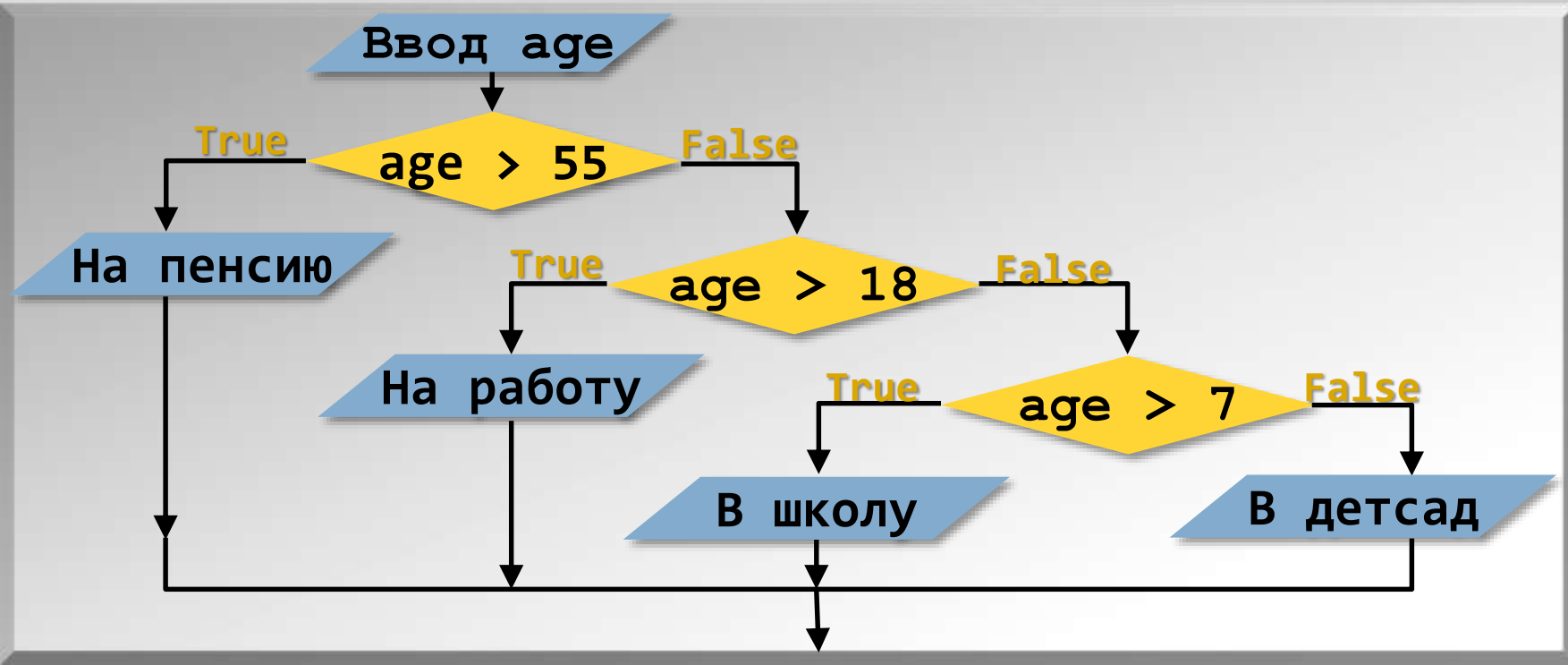



```
a = int(input('a = '))
b = int(input('b = '))
x = int(input('x = '))
if x>a and x<b:
    print('Да')
else:
    print('Нет')
```

40 Множественное Ветвление

```
if <условие1>:
    <блок инструкций1>
elif <условие2>:
    <блок инструкций2>
else:
    <блок инструкций3>
```

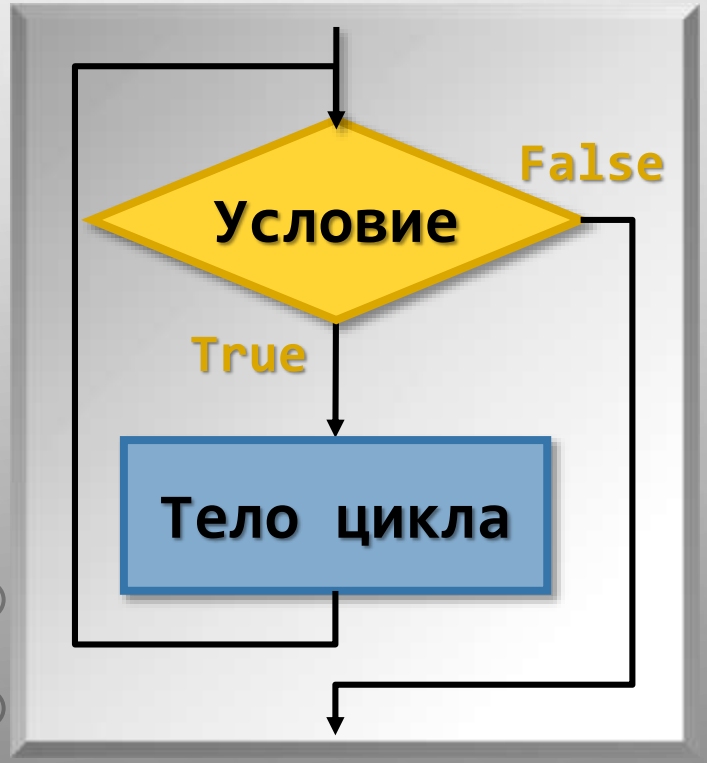
Если *условие1* истинно,
то будет выполнен *блок инструкций1*.
Если *условие1* ложно и *условие2* истинно,
то будет выполнен *блок инструкций2*.
Если *условие1* ложно и *условие2* ложно,
будет выполнен *блок инструкций3*.




```
age = int(input())
if age > 55:
    print('На пенсию')
elif age > 18:
    print('На работу')
elif age > 7:
    print('В школу')
else:
    print('В детсад')
```

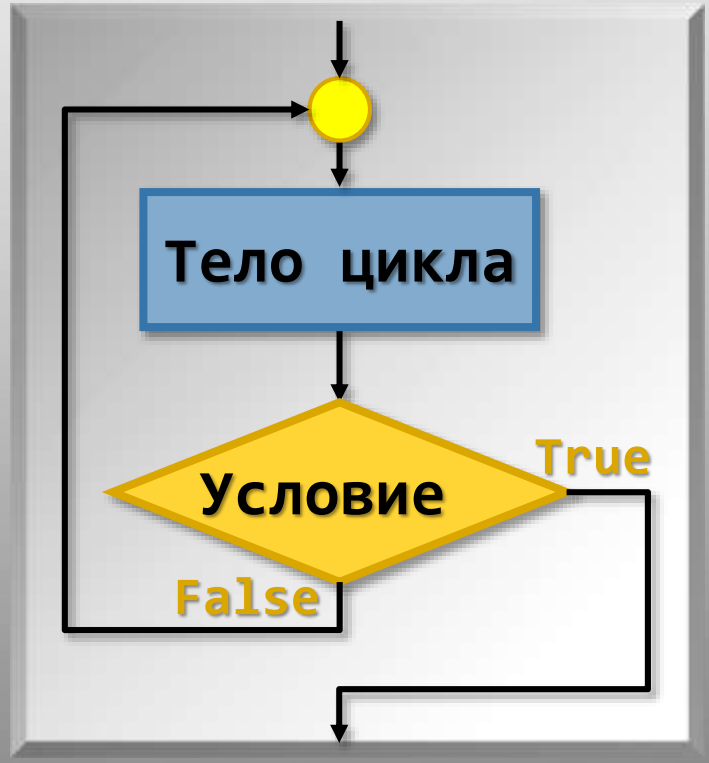
Циклы

Цикл с предусловием



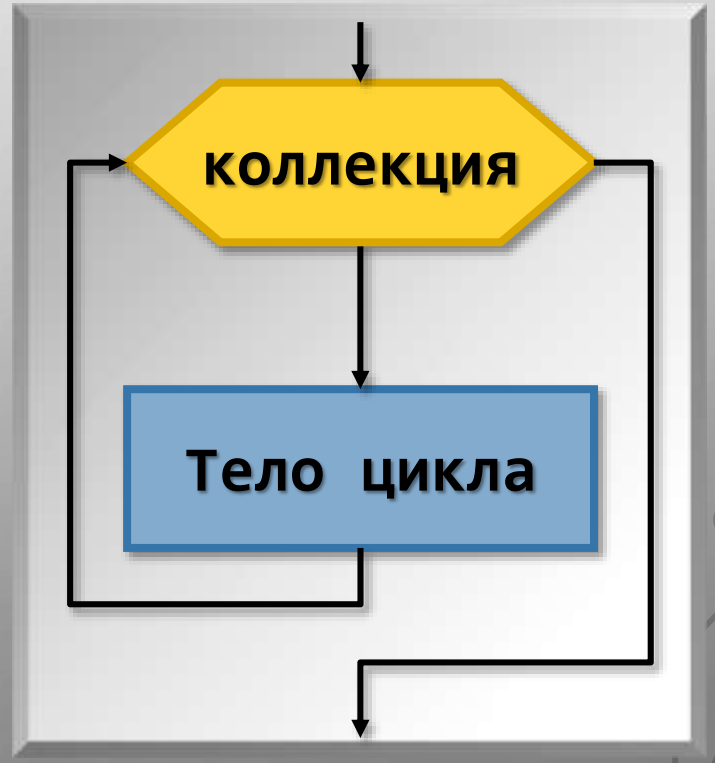
while

Цикл с постусловием



while break

Перебор коллекции



for in

Цикл с предусловием. WHILE

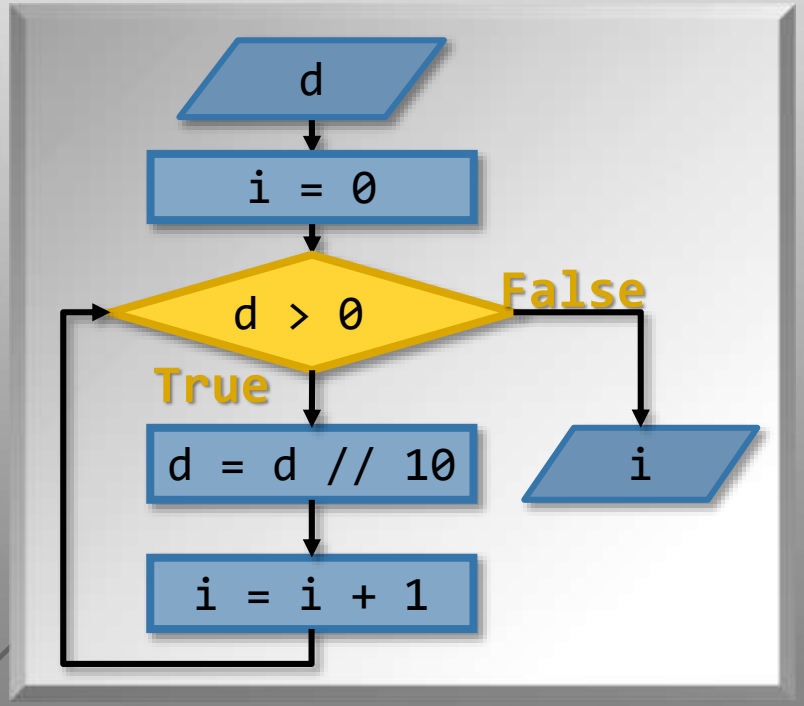
```
while <условие>:
    <блок инструкций1>
```

Синтаксис инструкции цикла с предусловием

Пока ***условие*** истинно,
будет выполняться ***блок инструкций1***
Когда ***условие*** станет ложно, цикл завершит работу.

Пример задачи

Определить количество цифр в десятичной записи целого числа.




```
d = int(input('d = '))
i = 0
while d > 0:
    d = d // 10
    i = i + 1
print(i)
```

Цикл с постусловием. WHILE

```
while <условие1>:
    <блок инструкций1>
    if <условие2>:
        break
    if <условие3>:
        continue
    <блок инструкций2>
else:
    <блок инструкций3>
```

Синтаксис инструкции цикла с постусловием

Пока ***условие1*** истинно,

будут выполняться ***блок инструкций1*** и ***2***, а также проверяться ***условие2*** и ***3.***

Если ***условие2*** истинно,

будет выполнен выход из цикла **break**, при этом ***блок инструкций3*** после оператор **else** выполнен не будет

Если ***условие3*** истинно,

будет выполнен переход на следующую итерацию цикла **continue**, при этом ***блок инструкций2*** выполнен не будет

Если ***условие1*** ложно,

будет выполнен ***блок инструкций2***, и цикл закончит работу.

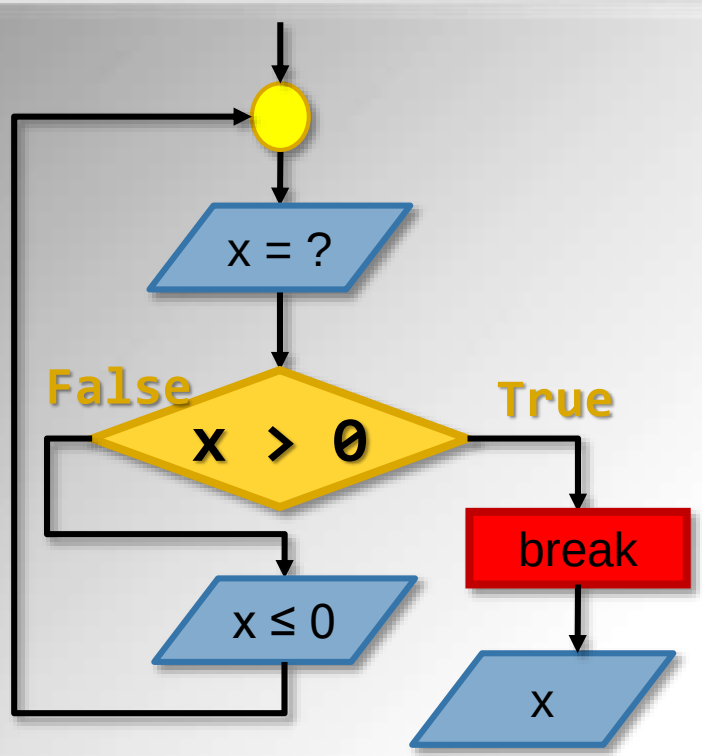
4 4 Цикл с постусловием. WHILE

Операторы прерывания цикла

break – оператор прерывания цикла, выполняющий выход из него

continue – оператор прерывания цикла, выполняющий переход на следующую итерацию

else – заголовок блока операций, которые будут выполнены один раз после окончания цикла, когда проверяемое условие станет неверно.



Пример задачи

Обеспечить ввод положительного числа в переменную x

```
while True:
    x = int(input('x = '))
    if x > 0:
        break
    print(x, '<= 0')
print('OK! X =', x)
```

Итерация1	Итерация2
-3	5
False	True
-3 <= 0	
	OK! X = 5

45 Перебор коллекции. For ... in ...

```
for <переменная> in <коллекция>:
    <блок инструкций1>
else:
    <блок инструкций2>
```

Синтаксис инструкции цикла с предусловием

Цикл перебирает каждый элемент ***коллекции***. Для каждого элемента проводится итерация цикла с выполнением ***блока инструкций1***, обратиться к текущему элементу коллекции можно по имени ***переменной***.

Цикл завершает работу после перебора каждого значения, либо после выполнения инструкции **break**.

Если цикл перебрал все элементы коллекции, то будет выполнен ***блок инструкций2***.

Цикл завершает работу

Пример задачи № 1

Вывести на экран степени двойки от 1 до n

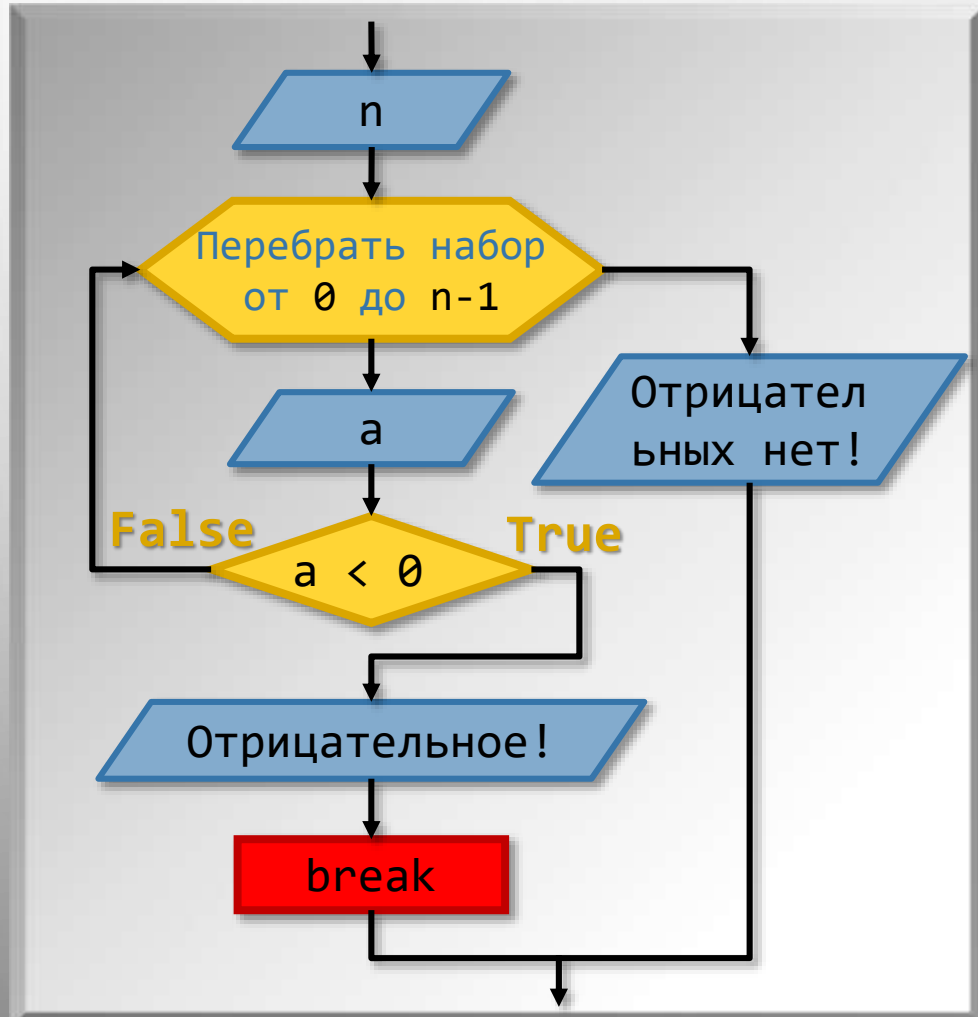
```
n = int(input('n = '))
for i in range(1, n+1):
    print(2**i)
```

Пример задачи № 2

Вывести на экран каждое слово из списка, с указанием количества букв

```
words = ['i','love','the','informatics']
for word in words:
    print(word, len(word))
```

Перебор коллекции. For ... in ...



Пример задачи № 3

Определить, есть ли среди **n**-вводимых с клавиатуры чисел отрицательные.

```
n = int(input('n = '))
for _ in range(n):
    a = int(input('число: '))
    if a < 0:
        print('Отрицательное!')
        break
else:
    print('Отрицательных нет!')
```

Последовательность чисел. range

```
range([start,] stop [, step])
```

Возвращает неизменяемую последовательность цифр

[start; stop) с шагом step

start – значение начала последовательности (по умолчанию 0).

stop – значение конца последовательности (обязательный параметр).

step – значение шага последовательности (по умолчанию 1).

```
range(8)
range(1,8)
range(1,0)
range(0,20,5)
range(0,-7,-1)
```

```
[0,1,2,3,4,5,6,7]
[1,2,3,4,5,6,7]
[]
[0,5,10,15]
[0,-1,-2,-3,-4,-5,-6]
```

48 Строки. str

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]
П	р	и	в	е	т	!
s[-7]	s[-6]	s[-5]	s[-4]	s[-3]	s[-2]	s[-1]

```
s = 'Привет!'
print(s)
print(len(s))
print(s + ' Ребята')
print(s * 3)
print(s[5])
print(s[-2])
print(s[len(s) - 2])
print(str(123) * 3)
s[6] = '.'
```

```
Привет!
7
Привет! Ребята
Привет!Привет!Привет!
т
т
т
123123123
Error
```

Строка – это неизменяемый объект!

Строки, срезы строк

s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]	s[8]	s[9]
a	b	c	d	e	f	g	h	i	j
s[-10]	s[-9]	s[-8]	s[-7]	s[-6]	s[-5]	s[-4]	s[-3]	s[-2]	s[-1]

```
st = 'abcdefghij'
s1 = st[3:8]
s2 = st[:8]
s3 = st[3:]
s4 = st[:]
s5 = st[1:-1]
s6 = st[::2]
s7 = st[::-1]
s8 = st[::-2]
```

```
abcdefghij
abcdefghij
abcdefghij
abcdefghij
abcdefghij
abcdefghij
jihgfedcba
jihgfedcba
```

```
defgh
abcdefgh
defghij
abcdefghij
bcdefghi
bdfhj
jihgfedcba
jhfdb
```

Строки, функции

Синтаксис	Описание
Поиск	
`i = len(st)`	Возвращает количество символов в переданной строке **st**.
`i = st.find(sub[, start[, end]])`	Возвращает относительную позицию первого вхождения подстроки **sub** в строку **st**, в пределах от начала **start** и до конца **end**. Если подстрока не найдена, то возвращает значение **-1**.
`i = st.count(sub[, start[, end]])`	Подсчитывает количество неперекрывающихся вхождений подстроки **sub** в строку **st**, в пределах от начала **start** и до конца **end**.
Разделение и соединение	
`li = st.split([sep[, max_count]])`	Возвращает список слов **li**, полученный из символьной строки **st**, используя разделитель **sep**, для её разделения. Строка разделяется не более **max_count** числа раз.
`st = sub.join(itr)`	Сцепляет итерируемый объект **itr** в единую строку **st**, причём строка **sub** вводится между элементами этого объекта.
`st_result = st.replace(sub, repl[, max_count])`	Возвращает строку **st_result** полученную из строки **st**, в которой выполняется замена всех вхождений подстроки **sub** на подстроку **repl**. Выполняется не более **max_count** замен.
Изменение	
`st = str(object)`	Преобразует объект **object** в строковый тип **st_result**.
`st_result = st.upper()`	Возвращает строку **st_result**, полученную после преобразования строчных буквы строки **st** в прописные.
`st_result = st.lower()`	Возвращает строку **st_result**, полученную после преобразования прописных буквы строки **st** в строчные.
`st_result = st.strip(simbols)`	Возвращает строку **st_result**, полученную из строки **st**, после удаления начальных и конечных символов **simbols**.

51 Регулярные выражения

Регулярное выражение – это специальная последовательность символов, определяющая шаблон, используемый для нахождения подходящих фрагментов в строке.
Используется для ***поиска***, ***разбиения*** на подстроки или ***замены*** фрагментов.

```
import re
```

Синтаксис	Описание
Функции модуля	
`match = re.match(pattern, st)`	Поиск первого совпадения **match** по шаблону **pattern** в начале строки **st**.
`match = re.search(pattern, st)`	Поиск первого совпадения **match** по шаблону **pattern** во всей строке **st**.
`lst = re.findall(pattern, st)`	Поиск всех совпадений **lst** по шаблону **pattern** во всей строке **st**.
`lst = re.split(pattern, st)`	Разделение строки **st** по шаблону **pattern** на части (левее и правее от совпадения). Возвращает список **lst**.
`st = re.sub(pattern, repl, st)`	Замена всех совпадений с шаблоном **pattern** на фрагмент **repl**, в строке **st**.
`regex = re.compile(pattern)`	Создание объекта регулярного выражения **regex** из шаблона **pattern**, который можно использовать в дальнейшем.
Объекты совпадений	
`st|tupl = match.group([group_N,…])`	Возвращает одно или более найденных совпадений, • по умолчанию возвращает все совпадения в виде строки **st**, • при указании номера совпадения возвращает только его в виде строки, • при указании нескольких возвращает кортеж **tupl**, состоящий из строк-совпадений.
`tupl = match.groups()`	Возвращает кортеж **tupl**, содержащий все найденные совпадения.
`i = match.start([group_N])`	Возвращает индекс **i** левой границы найденного совпадения с номером **group_N** (по умолчанию первого).
`i = match.end([group_N])`	Как и **start**, но возвращает правую границу найденного совпадения.
`tupl = match.span([group_N])`	Возвращает кортеж **tuple**, состоящий из левой и правой границы найденного совпадения.

Регулярные выражения

Специальные символы

`.`	1 любой символ, кроме новой строки «\n»
`\w \W`	Любая цифра или буква | Всё кроме
`\d \D`	Любая цифра | Всё кроме цифры
`\s \S`	Любой пробельный символ | Всё кроме
`\`	Экранирует специальные символы

Количество элементов

`?`	0 или 1 вхождение шаблона слева
`+`	1 или более вхождений шаблона слева
`*`	0 или более вхождений шаблона слева
`{n,m}`	От **n** до **m** вхождений

Группировка

`(регвыр)`	Выделяет группу
`(?P<имя>регвыр)`	Выделяет именованную группу

Расположение

`^`	Поиск в начале строки
`$`	Поиск в конце строки
`\b \B`	Граница слова | Всё кроме границы

Наборы для сравнения

`a|b`	Один из двух символов, **a** или **b**
`[a-zA-Z]`	Один из символов в скобках
`[^3-7]`	Любой символ, кроме тех, что в скобках

Списки, list

ИМЯ списка

li

ИНДЕКС элемента списка: 0

ЗНАЧЕНИЕ элемента массива: 3

li[0]	li[1]	li[2]	li[3]	li[4]	li[5]	li[6]	li[7]	li[8]	li[9]
12	3	'a'	-1	4.3	5	33	0	3	2
li[-10]	li[-9]	li[-8]	li[-7]	li[-6]	li[-5]	li[-4]	li[-3]	li[-2]	li[-1]

```
li[2]      'a'
li[0]      12
li[-2]     3
li[1:3]    3, 'a'
li[:6:2]   12, 'a', 4.3
li[5::2]   5, 0, 2
```

Список – это массив ссылок

Список – это упорядоченная последовательность объектов произвольного типа.

Характеристики:

- Упорядоченный (*индексы от 0,1,2,3,4…*)
- Изменяемый тип (*в отличии от строк*)
- Расширяемый (*по необходимости, без создания копий*)
- Гетерогенный (*т.е. содержащий различные типы данных*)
- Итерируемый (*т.е. перечисляемый*)

Списки, создание

```
li1 = [1, 3, 4, 23, 5]
li2 = [1, 3] + [4, 23]
li3 = [0] * 9
li4 = []
```

```
[1, 3, 4, 23, 5]
[1, 3, 4, 23]
[0, 0, 0, 0, 0, 0, 0, 0, 0]
[]
```

Создание с помощью встроенных функций

```
li5 = list(range(1,10))
li6 = list('abcde')
li7 = '1 3 4 23 5'.split()
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
['a','b','c','d','e']
['1','3','4','23','5']
```

Генераторы списков

```
li8 = [i**2  for i in range(1,7)]
li9 = [i for i in range(10) if i%2]
```

```
[1, 4, 9, 16, 25, 36]
[1, 3, 5, 7, 9]
```

Списки, ввод с клавиатуры

Ввод в одну строку

```
li1 = [int(i) for i in input().split()]
li1 = list(map(int, input().split()))
```

```
>>> 1 3 4 23 5
[1, 3, 4, 23, 5]
```

Ввод с новой строки

```
N = int(input())
li2 = [0] * N
for i in range(N):
    li2[i] = int(input())
```

```
N = int( input() )
li2 = [int(input()) for i in range(N)]
```

```
N = int(input())
li2 = []
for i in range(N):
    li2.append(int(input()))
```

```
N = int(input())
li2 = [0] * N
for i in range(len(li2)):
    li2[i] = int(input())
```

```
>>> 3
1
3
4
[1, 3, 4]
```

Списки, Генерация случайных значений

```
from random import randint
```

Цикл с предусловием

```
N = int(input())
i = 0
li = [0] * N
while i < N:
    li[i] = randint(20, 100)
    i += 1
```

Цикл с перебором коллекции

```
N = int(input())
li = [0] * N
for i in range(N):
    li[i] = randint(20, 100)
```

Генераторы списков

```
N = int(input())
A = [randint(20, 100) for x in range(N)]
```

Списки, вывод на экран

li	1	2	3	4	5

Как список:

```
print(li)
```

```
[1, 2, 3, 4, 5]
```

В строчку через пробел:

```
for i in range(N):
    print(li[i], end=' ')
```

```
1 2 3 4 5
```

или так:

```
for x in li:
    print(x, end=' ')
```

```
1 2 3 4 5
```

быстрее всего так:

```
print(' '.join([str(i) for i in li]))
```

```
1 2 3 4 5
```

Двухмерные списки. Создание

Матрица 4x3

2	56	-5	2
0	46	3	10
88	5	7	-8

Создание из определённых значений:

```
M = [[2,56,-5,2],[0,46,3,10],[88,5,7,-2]]
```

```
>>> M[1]
[0,46,3,10]
```

```
>>> M[0][3]
2
```

Заполнение матрицы значениями с клавиатуры:

```
m,n = 3,4
M = [0] * m
for i in range(m):
    M[i] = [0]*n
    for j in range(n):
        M[i][j] = int(input())
```

или

```
m = 4
M = []
for i in range(m):
    row = list(map(int, input().split()))
# либо так:
# row = [int(j) for j in input().split()]
    M.append(row)
```

```
n = 4
M = [[int(j) for j in input().split()] for i in range(n)]
```

Двухмерные списки. Создание

Заполнение матрицы случайными значениями:

```
from random import randint
m, n = 4, 3
M = [0] * m
for i in range(m):
    M[i] = [0]*n
    for j in range(n):
        M[i][j] = randint(0,10)
```

```
from random import randint
m, n = 4, 3
M = []
for i in range(m):
    M.append([])
    for j in range(n):
        M[i].append(randint(0,10))
```

ИЛИ

```
from random import randint
m, n = 4, 3
M = [[randint(0,10) for j in range(n)] for i in range(m)]
```

Двухмерные списки. Вывод

Вывод матрицы в нескольких строках:

```
for i in range(len(M)):
      for j in range(len(M[i])):
            print(M[i][j], end=' ')
      print()
```

или

```
for row in M:
      for elem in row:
            print(elem, end=' ')
      print()
```

или

```
for row in M:
      print(' '.join([str(elem) for elem in row]))
```

Матрица 4x3

2	56	-5	2
0	46	3	10
88	5	7	-8

```
2 56 -5 2
0 46 3 10
88 5 7 -8
```

Списки, функции

Функция	Примеры	Описание
Создание		

L = list(I)	>>> L = list(range(10))
	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
	>>> L1 = list('hello')
	['h', 'e', 'l', 'l', 'o']
		Возвращает список *L*, созданный из итерируемого объекта

L = S.split(S1, X)	>>> L2 = '1 d 23 s'.split()
	[1, 'd', 23, 's']
	>>> L3 = '192.168.0.1'.split('.')
	['192', '168', '0', '1']
		Возвращает список слов *L* в символьной строке *S*, используя разделитель *S1* (по умолчанию пробел), для их разделения. Если задан параметр *X*, то строка разделяется не меньше этого числа раз.

S = S1.join(I)	S = ' '.join(L) '1 2 3 4 5 6 7 8 9'	Сцепляет итерируемый объект *I* в единую строку *S*, причём строка *S1* вводится между элементами этого объекта.

Списки, функции

Функция	Примеры	Описание
Добавление и удаление элементов		

L.append(X)
или
L[len(L):] = [X]
	>>> L.append(10)
	>>> L[len(L):] = [10]
	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		Вставляет одиночный объект ***X*** в конец списка ***L***, непосредственно изменяя список

L.insert(i,X)
или
L[i:i] = [X]
	>>> L.insert(5,777)
	>>> L[5:5] = [777]
	[0, 1, 2, 3, 4, 777, 5, 6, 7, 8, 9, 10]
		Вставляет одиночный объект ***X*** в список ***L*** по индексу ***i***.

L.extend(I)
или
L[len(L):] = I
	>>> L.extend('abc')
	>>> L[len(L):] = 'abc'
	[0, 1, 2, 3, 4, 777, 5, 6, 7, 8, 9, 10, 'a', 'b', 'c']
		Вставляет каждый элемент из итерируемого объекта ***I*** в конец списка ***L***.

del L[l,j]	>>> del L[-3:] [0, 1, 2, 3, 4, 777, 5, 6, 7, 8, 9, 10]	Удаляет элементы списка ***L*** с указанными индексами

L.remove(X)
или
del L[L.index(X)]
	>>> L.remove(777)
	>>> del L[L.index(777)]
	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
		Удаляет первое вхождение объекта ***X*** из списка ***L***. Генерирует исключение, если этот объект не найден в списке.

X = L.pop(i)
или
X = L[i]; del L[i];
	>>> X = L.pop()
	>>> X = L[-1]; del L[-1];
	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
		Возвращает последний элемент ***X*** и удаляет его из списка ***L*** (или же элемент по индексу ***i***)

Списки, функции

Функция	Примеры	Описание
Поиск, анализ		
`X = len(L)`	`>>> X = len(L)` `10`	Возвращает количество элементов *X* в списке *L*

`k = L.index(X,i,j)`	`>>> k = L.index(4)`
	`5`
	`>>> L4 = [1,2,3,3,3,2,1]`
	`>>> k = L4.index(3,4,6)`
	`4`
		Возвращает индекс *k* первого вхождения объекта *X* вы список *L*. Генерирует исключение, если этот объект не найден в списке. Если ему передаётся параметр *i*, и *j*, то он возвращает наименьший индекс *k*, чтобы *L*[*k*] == *X* and *i* <= *k* < *j*, где *j* по умолчанию равно концу списка.

`k = L.count(X)`	`>>> X = L4.count(3)` `3`	Возвращает количество вхождений *k* объекта *X* в список *L*.
`X = min(L)`	`>>> X = min(L)` `0`	Возвращает значение минимального элемента *X* в списке *L*.
`X = max(L)`	`>>> X = max(L)` `9`	Возвращает значение максимального элемента *X* в списке *L*.
`X = sum(L)`	`>>> X = sum(L)` `45`	Возвращает сумму элементов *X* в списке *L*.
Изменение		
`L.reverse()`	`>>> L.reverse()` `[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]`	Обращает элементы непосредственно в списке *L*.

`L.sort([reverse=True|False])`	`>>> L.sort(reverse=True)`
	`[0,1,2,3,4,5,6,7,8,9]`
	`>>> L1 = [45, 11, 92, 3, 8]`
	`>>> L1.sort()`
	`[3, 8, 11, 45, 92]`
		Сортирует список *L*, непосредственно (по умолчанию это делается по возрастанию). В случае, если задан параметр reverse = True, то сортировка производится по убыванию.

Словари, dict

Словарь (ассоциативный массив, либо таблица хэшей) – это неупорядоченная коллекция пар «ключ-значение», индексируемая по ключам (строковым значениям), а не по номеру позиции (как в списках)

```
dic = {
 "GOOG":490.10, "AAPL":123.50,
 "IBM":91.50, "MSFT": 52.13}
```

Ключ	Значение
GOOG	490.10
AAPL	123.50
IBM	91.50
MSFT	52.13

Характеристики:

- Не упорядоченный (*элементы хранятся в неопределённом порядке*)
- Доступ по ключу (*на основе быстрой операции поиска*)
- Изменяемый тип (*в отличии от строк*)
- Расширяемый (*по необходимости, без создания копий*)
- Гетерогенный (*т.е. содержащий различные типы данных*)
- Итерируемый (*т.е. перечисляемый*)

Операции со словарями

```
dic = dict([("GOOG",490.10),("AAPL",123.50),
            ("IBM",91.50),("MSFT",52.13)])
```

Ключ	Значение
GOOG	490.10
AAPL	123.50
IBM	91.50
MSFT	52.13

```
len(dic)                    4
"IBM" in dic                True
91.50 in dic                False
dic["IBM"]                  91.50
dic["YNDX"]                 KeyError
dic.get("YNDX",0)           0
dic["YNDX"] = 19.13         "YNDX" in dic
del dic["MSFT"]             "MSFT" not in dic
el = dic.pop("FB",0)        el = 0
dic.update({"FB":117.74})   "FB" in dic
```

Перечисление элементов словаря

```
dic.keys()
dic.values()
dic.items()
```

```
['IBM','GOOG','AAPL','MSFT']
[91.5, 490.1, 123.5, 52.13]
('IBM', 91.5), ('GOOG', 490.1),
 ('AAPL', 123.5), ('MSFT', 52.13)
```

Ключ	Значение
GOOG	490.10
AAPL	123.50
IBM	91.50
MSFT	52.13

```
for a in dic:
    print(a, dic[a])
```

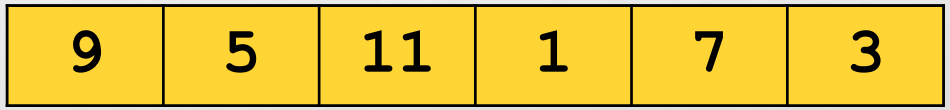
```
for key,val in dic.items():
    print(key,val)
```

```
IBM 91.5
GOOG 490.1
AAPL 123.5
MSFT 52.13
```

Множества, set

```
st = {1,1,3,5,7,9,11}
```

```
st = set(3,1,5,7,11,3,9)
```



Множество – это неупорядоченная коллекция уникальных и не изменяемых элементов, поддерживающая такие операции, как пересечение, объединение, разность и т.д.

Характеристики:

- Не упорядоченный (*элементы не имеют индекса*)
- Не повторяющийся (*элементы множества не повторяются*)
- Изменяемый тип (*в отличии от строк*)
- Расширяемый (*по необходимости, без создания копий*)
- Гетерогенный (*т.е. содержащий различные типы данных*)
- Итерируемый (*т.е. перечисляемый*)

Множества, set

```
A = {1,2,3,4,5}
```

```
B = {1,2,3}
```

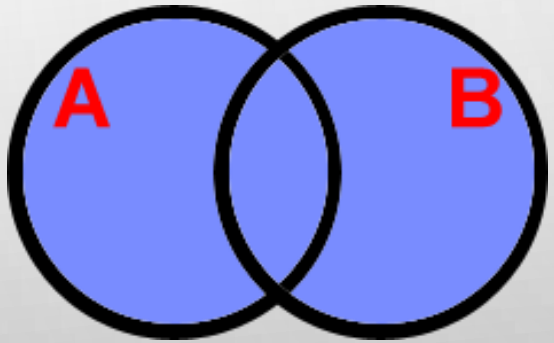
```
len(A)
1 in A
B in A
B <= A
B < A
A.add(6)
A.discard(2)
A.remove(7)
el = A.pop()
```

```
5
True
False
True
True
{1,2,3,4,5,6}
{1,3,4,5,6}
KeyError
el = 5, A = {1,3,4,6}
```

Множества, set

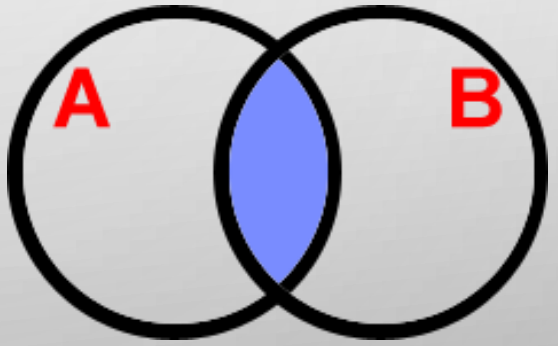
```
A = {1,2,3,4,5}
```

```
B = {4,5,6,7,8}
```



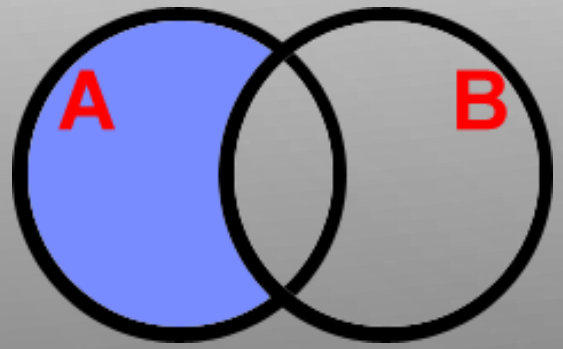

```
A | B
A |= B
```

{1,2,3,4,5,6,7,8}



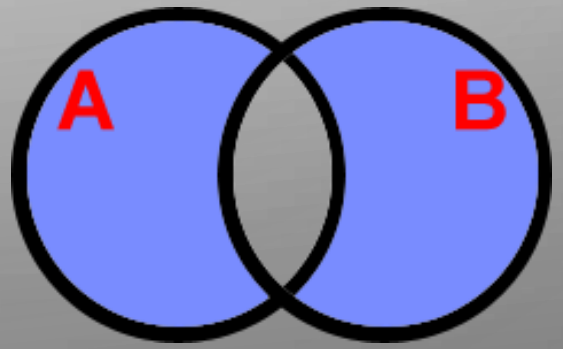

```
A & B
A &= B
```

{4,5}




```
A - B
A -= B
```

{1,2,3}




```
A ^ B
A ^= B
```

{1,2,3,6,7,8}

Файлы

Файлы делятся на 2 типа:

Текстовые файлы	данные файла автоматически декодируются при чтении в тип `str`, а при записи объекты `str` кодируются в данные (наборы байт).
Двоичные файлы	данные считываются и записываются без преобразования, представляя тип `bytes` (в описание режима добавляется символ `'b'`)

Строки делятся на 2 типа:

- `str` – для представления текстовых строк, состоящих из символов Юникода;
- `bytes` (`bytearray`) – для представления двоичных данных.

Чтение:
```
f = open('<имя файла>'[, '<режим>'[, encoding='<кодировка>']])
```

Запись:
```
print(<данные>, file='<имя файла>')
```

Режимы открытия файлов бывают 3 типов:

`r` **(read)** – для открытия на чтение (режим по умолчанию)

`w` **(write)** – для открытия на запись

`a` **(append)** – для открытия на добавление в конец

Самые распространённые кодировки:

`utf-8` – кодировка **Unicode**, с переменным числом байт на символ (по умолчанию). Русский язык [1040, 1103].

`cp1251` – 8-битная кодировка **Windows-1251** для 255 символов. Русский язык [192, 255].

`ascii` – стандартная 7-битная кодировка для 128 символов. Русского языка нет.

Операции с файлами

Синтаксис	Описание
`f = open('text.txt', 'r')`	
`strg = f.read(N)`	Чтение файла `f` целиком (либо `N` символов из него) в строку `strg`.
`strg = f.readline()`	Чтение 1 текущей строки из файла `f` в строку `strg`.
`lst = f.readlines()`	Чтение файла целиком в список строк `lst`
`f = open('text.txt', 'w')`	
`f.write(strg)`	Запись строки `strg` символов в файл `f`
`f.writelines(lst)`	Запись всех строк из списка `lst` в файл `f`
`f.seek(N)`	Изменение текущей позиции в файле, смещая её на `N` символов (или байт)
`f.close()`	Закрытие файла, высвобождение памяти

Хороший способ прочитать файл

```
for line in open('text.txt'):
    print(line)
```

Безопасный способ прочитать файл

```
with open('text.txt') as file:
    for line in file:
        print(line)
```

Хороший способ записать файл

```
data = ['a', 'b', 'c', 'd', 'e']
f = open('text.txt', 'w')
for element in data:
    print(element, file = f)
f.close()
```

Функции

1. Назначение, описание

Назначение, описание

Функции нужный для:

- Многократного использования участков кода (*т.е. минимизации избыточности кода*);
- Процедурной декомпозиции (*т.е. разделения задачи на более простые составляющие*).

Операторы для работы с функциями:

- **`def имя_функции:`** – исполняемый оператор, определяющий начало описания функции, и связывающий созданный объект-функцию с её именем.
- **`return объект`** – останавливает работу функции и передаёт **`объект`**, как результат своей работы, в вызвавшую её программу.
- **`global имя`** – связывает имя с объектом из глобальной области, имеющим такое же имя.
- **`nonlocal имя`** – связывает имя с объектом из объемлющей области, имеющим такое же имя (имя должно существовать в момент создания!).

Пространства видимости имён

Функции

Глобальные переменные

```
def Func(a,b):
    a = a + 8
    c = 3
    d += 1
    global e; e = 6
    return a+b+c+d+e
a, d, e = 1, 4, 5
Func(a,2)
a
b
c
d
e
```

```
25
1
Error
Error
4
6
```

ПЕРЕМЕННЫЕ	a	b	c	d	e
Глобальные	1	-	-	4	6
Локальные	9	2	3	5	

Фабричная функция

```
def tester(start):
    state = start
    def nested(label):
        nonlocal state
        print(label, state)
        state += 1
    return nested
F = tester(0)
F('spam')
F('ham')
F('eggs')
G = tester(10)
G('spaghetti')
```

```
spam 0
ham 1
eggs 2

spaghetti 10
```

Функции(аргументы)

Функции нужный для:

- При передаче аргументов в функцию происходит передача ссылок на объекты (а не копирование объектов).

```
def Func(x,y,z):
    x = 2
    y[0] = 'i'
    z[1] = 'j'
a = 1
b = [1,2]
c = [3,4]
Func(a,b,c[:])
a
b
c
```

```
1
['a',2]
[3,4]
```

Синтаксис	Описание
Вызов функции	
Func(value)	Обычный аргумент(позиционный) – сопоставление по позиции
Func(name=value)	Именованный аргумент – сопоставление по имени
Func(*sequence)	Распаковка последовательности в отдельные позиционные аргументы
Func(**dict)	Распаковка словаря в отдельные именованные аргументы
Описание функции	
def Func(name):	Обычный аргумент – сопоставление по позиции
def Func(name=value):	Значение аргумента по умолчанию, на случай, если аргумент не передан
def Func(*name):	Объединяет множество обычных аргументов в кортеж
def Func(**name):	Объединяет множество именованных аргументов в словарь

Функции(аргументы)

```
def f(a,b,c):
    print(a,b,c)
f(1,2,3)
f(c=3,b=2,a=1)
f(1,c=3,b=2)
```

```
1 2 3
1 2 3
1 2 3
```

```
def f(a,b=2,c=3):
    print(a,b,c)
f(1)
f(a=1)
f(1,4)
f(1,4,5)
f(1,c=6)
```

```
1 2 3
1 2 3
1 4 3
1 4 5
1 2 6
```

```
def f(*args):
    print(args)
f()
f(1)
f(1,2,3,4)
```

```
()
(1,)
(1,2,3,4)
```

```
def f(a,b,c):
    print(a,b,c)
L = [1,2,3]
f(L)
f(*L)
```

```
ОШИБКА!
1 2 3
```

Трассировочная функция

```
def tracer(func,*pargs,**kargs):
    print('Вызов:',func.__name__)
    return func(*pargs, **kargs)
def func(a,b,c,d):
    return a + b + c + d
print(tracer(func,1,2,c=3,d=4))
```

```
Вызов: func 10
```

Функции и рекурсия

Рекурсия:

- Реализует способ вычисления очередного значения функции через вычисление её предшествующих значений.
- Основана на применении стека.
- Полезна для обхода структур с произвольной организацией (деревья, цепочки, и т.д.)

Последовательность Фиббоначи

$$f(n) = f(n-1) + f(n-2),\ n >= 3.$$

```
def fib(n):
    if n==1 or n==2:
        return 1
    else:
        return fib(n-1)+fib(n-2)
```

Сумма элементов

```
def rec_sum(L):
    if not L:
        return 0
    else:
        return L[0]+rec_sum(L[1:])
```

Дополнительно о функциях

Атрибуты функций:

- Функция это объект, который можно именовать и передавать.
- К функциям можно прикреплять атрибуты, которые хранится в словаре __dict__

```
def echo(mes): print(mes)
echo.comm = 'abc'
echo.count = 0
echo('Hello')
echo.count += 1
echo.count; echo.comm
```

```
Hello

1 abc
```

Подсчёт количества вызовов

```
def mysum(*args):
  if 'calls' in mysum.__dict__:
    mysum.__dict__['calls'] += 1
  else:
    mysum.__dict__['calls'] = 0
  print('Calls №', mysum.calls)
  return sum(args)
```

Дополнительно о функциях

lambda АРГУМЕНТЫ,… : ВЫРАЖЕНИЕ

lambda-функции:

- В отличии от **def**, не именует, а возвращает функцию (которую, кстати, можно присвоить в переменную)
- Это не блок инструкций, а выражение!
- Циклы можно заменить использованием **lambda** вместе с **map**

Скорость C

```
lambda x: x**2
f = lambda x,y: x*y
f(7*8)
L = [1,2,3]
list( map(lambda x: x+10, L) )
```

Функциональное программирование

Описание:

- Программа представляет собой вычисление значений функций (в матем. смысле), которые по исходным данным получают результат. Без изменения состояния переменных (как в императивном).
- Для реализации применяется: рекурсия, “фабричные” функции, функции **map**, **filter**, **reduce**, инструкция **lambda**, генераторы…

```
def sumDigits(x):
    if x > 0:
        return x % 10 + \
            sumDigits(x//10)
    else: return 0
```

```
def isPalindrome(s):
    if len(s) > 1:
    return s[0] == s[-1] and \
        isPalindrome(s[1:-1])
    else: return True
```

```
L = [0]*10
L = map( lambda x: randint(0,100), L )
L1 = filter( lambda x: x % 3 == 0, L )
L2 = map( lambda x: x**2, L1 )
L3 = reduce( lambda p, x: p*x, L2 )
```

Функции	Описание
`Mp = map(func, It)`	Возвращает итератор *Mp*, созданный применением функции *func* к итератору *It*. Количество параметров *N*, которые принимает *func* равно количеству переданных итераторов *It1*, *It2*, … ,*ItN*.
`Ft = filter(func, It)`	Возвращает итератор *Ft*, созданный из элементов итератора *It*, для которых функция *func* дала результат `True`.
`Rd = reduce(func, It)`	Применяет функцию *func* к каждому элементу последовательности *It*, накапливая результат.

Итераторы

Функции	Описание
`It = iter(ItOb)`	Возвращает итератор *It* итерируемого объекта *ItOb*
`El = next(It)`	Возвращает следующий элемент *El* итератора *It*, когда элементы заканчиваются – генерирует исключение *StopIteration*

Итерируемые объекты:

- Строки, списки, кортежи, файлы;
- Множества, словари (в т.ч. функции словарей keys(), values(), items());
- Функции range(), sorted();
- Функции map(), zip(), filter(), enumerate() – сами являются итераторами (т.е. позволяют сделать только 1 проход по значениям).

```
L = [1,2,3]
Li = iter(L)
Li is L
next(Li); next(Li); next(Li);
next(Li);
```

```
[1,2,3]
ИТЕРАТОР
False
1 2 3
StopIteration ИСКЛЮЧЕНЕ
```

Итераторы

Итерируемые объекты – это объекты, которые могут возвращать итераторы, т.е. имеют метод **__iter__()**.

Итераторы – это объекты, которые по требованию (вызов **next()**) возвращают очередной элемент последовательности, т.е. содержат метод **__next__()**, либо используют конструкцию **yield**.

```
class Colors2:
    def __init__(self, colors):
        self.colors = colors
        self.i = 0

    def __iter__(self):
        return self

    # def next(self)
    #     pass

    def __next__(self):
        self.i += 1
        if self.i == len(self.colors):
            self.i = 0
        return self.colors[self.i]
```

```
class Colors:
    def __init__(self, colors):
        self.colors = colors

    def __iter__(self):
        i = 0
        while True:
            yield self.colors[i]
            i += 1
            if i == len(self.colors):
                i = 0
```

```
import random

class RandNumbers:
    num = 0

    def __iter__(self):
        return self

    def __next__(self):
        self.num += 1
        if self.num > 100:
            raise StopIteration
        return random.randint(-10, 10)
```

```
import itertools
# itertools.chain() - позволяющий объединить два разных итератора в о
# itertools.count() -выдающий бесконечные числа, начиная с заданного
# itertools.cycle() - Бесконечно повторяет заданную последовательност
```

Функции для итераторов

Функции	Пример	Описание

`S = sum(It)`	`>>> It = iter(range(10))`
	`>>> sum(It)`
	`45`
		Возвращает сумму *S* элементов итератора *It*

`M = max(It)`
`M = min(It)`
	`>>> It = iter(range(10))`
	`>>> max(It), min(It)`
	`(9, 0)`
		Возвращает максимальный (минимальный) элемент *M* итератора *It*

`Mp = map(func, It)`	`>>> f = lambda x: x**2`
	`>>> list( map(f, range(6)) )`
	`0 1 4 9 16 25`
		Возвращает итератор *Mp*, созданный применением функции *func* к итератору *It*. Количество параметров, которые принимает *func* равно количеству переданных итераторов.

`Sr = sorted(It)`	`>>> It = iter('dbca')`
	`sorted(It)`
	`['a','b','c','d']`
		Возвращает отсортированный список *Sr*, созданный из итератора *It*.

`En = enumerate(It)`	`>>> It = iter('abc')`
	`>>> En = enumerate(It)`
	`>>> list(En)`
	`[(0, 'a'), (1, 'b'), (2, 'c')]`
		Возвращает итератор *En*, созданный из итератора *It*, добавлением номера позиции каждому элементу итератора *It*.

`Zp = zip(It1, It2)`	`>>> It1 = iter( range(3) )`
	`>>> It2 = iter('abc')`
	`>>> Zp = zip(It1, It2)`
	`>>> list(Zp)`
	`[(0, 'a'), (1, 'b'), (2, 'c')]`
		Возвращает итератор *Zp*, созданный объединением значений итераторов *It1* и *It2*, имеющих одинаковый индекс.

Генераторы

- **Генераторы** – это близкие родственники цикла **for**, позволяющие применять *выражения* и *условия* к *итерируемой последовательности*, и возвращающие *список*.
- **Выражение-генератор** это – если *генератор* заключить в круглые скобки (он вернёт *итератор*).
- **Функция-генератор** – похожа но обычную функцию **def**, но содержит **yield**, которая возвращает значения по одному, приостанавливая выполнение.

```
X = [ ВЫРАЖЕНИЕ for ПЕРЕМЕННАЯ in ИТЕРИРУЕМЫЙ if УСЛОВИЕ ]
      ФУНКЦИЯ          ЦИКЛА          ОБЪЕКТ
```

```
L = [1,2,3,4,5]
L1 = [x+10 for x in L if x%2]
```

Скорость C

```
[1,2,3,4,5]
[11,13,15]
```

```
L1 = []
for x in L:
    if x%2:
        L1.append(x+10)
```

БОЛЕЕ МЕДЛЕННЫЙ АНАЛОГ
Скорость Python

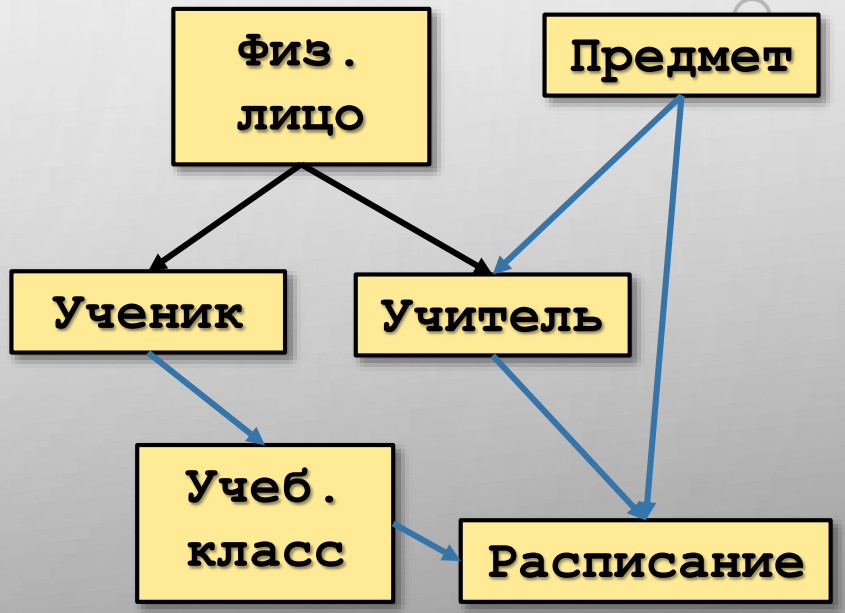
```
[11,13,15]
```

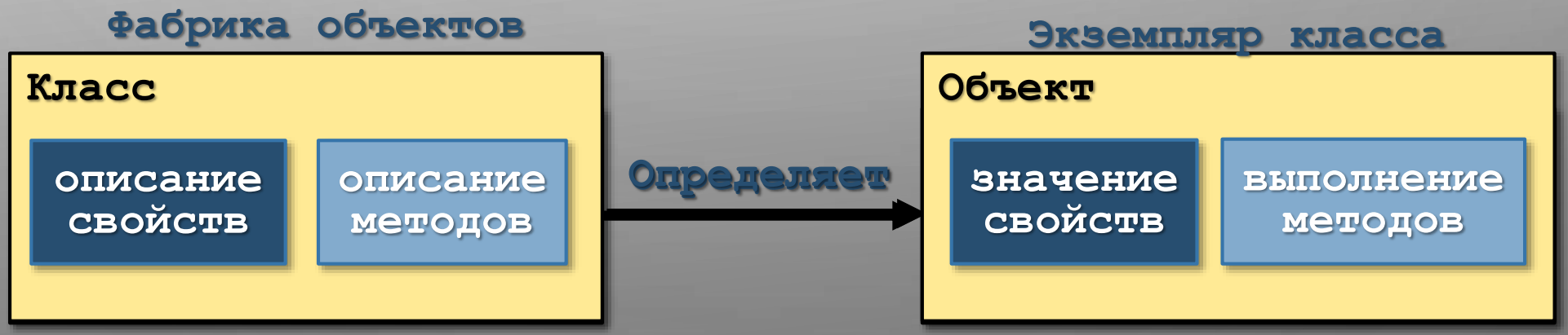
Объектно-ориентированное программирование (ООП)

1. Основные термины
2. Пространства имён
3. Описание классов
4. Инкапсуляция и интерфейс
5. Наследование. пример 1
6. Обход дерева наследования
7. Специализация наследования
8. UML диаграммы классов
9. Шаблоны проектирования
10. Перегрузка операторов

ООП. Основные термины

- **Абстракция** – это выделение существенных свойств объекта, отличающих его от других объектов (в рамках конкретной задачи).
- **Декомпозиция** – это разделение объекта на составляющие элементы (подобъекты).
- **Объект** – это то, что имеет чёткие границы, обладает состоянием и поведением.
- **Класс** – это множество объектов, имеющих общую структуру и поведение.
- **Состояние (свойство)** – это переменная, принадлежащая объекту.
- **Поведение (метод)** – это функция, принадлежащая классу объектов
- **Наследование** – это передача свойств и методов класса – классам-наследникам.





ООП. Пространства имён

Дерево наследования объектов

- **Объект** – это пространство имён.
- **Наследование** имён – происходит **сверху-вниз**.
- **Поиск** имён – происходит **снизу-вверх.**

Данные класса и данные экземпляра

```
class ShareData:
  spam = 42 # атрибут данных класса
  def __init__(self, value=0):
     self.data = value
     # атрибут экземпляра

x = ShareData(123) # создаём экземпляры
y = ShareData()

print(x.spam, y.spam)                    42 42
print(x.data, y.data)                    123 0
ShareData.spam = 13
x.Spam = 88 # атрибут данных экземпляра
print(x.spam, y.spam, ShareData.spam)    88 13 13
```

Методы экземпляра и класса

```
class NextClass:
  def printer(self, text):
     self.data = text
     print(self.data)

x = NextClass()
x.printer('Hello')                  Hello
print(x.data)                       Hello
NextClass.printer(x,'Goodbye')      Goodbye
# прямой вызов метода
print(x.data)                       Goodbye
```

ООП. Пространства имён

Всё о пространствах имён

```
x = 11 # глобальный атрибут модуля

def f():
    print(x) # обращение к
             # глобальному атрибуту
def g():
    x = 22   # локальная переменная
    print(x) # функции g()

class C:
    x = 33          # атрибут класса C
    def m(self):
        x = 44      # локальная переменная
                    # функции m() класса C
        self.x = 55 # атрибут экземпляра
                    # класса C
```

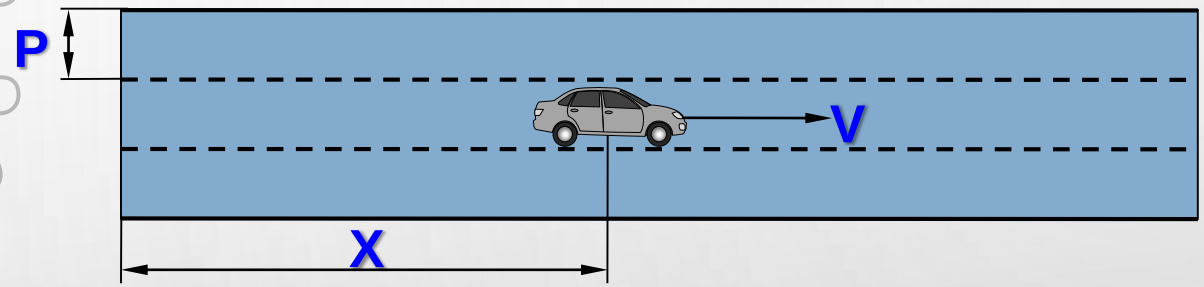
```
print(x)
f()
g()
print(C.x)
obj = C()
print(obj.x)
obj.m()
print(obj.x)
```

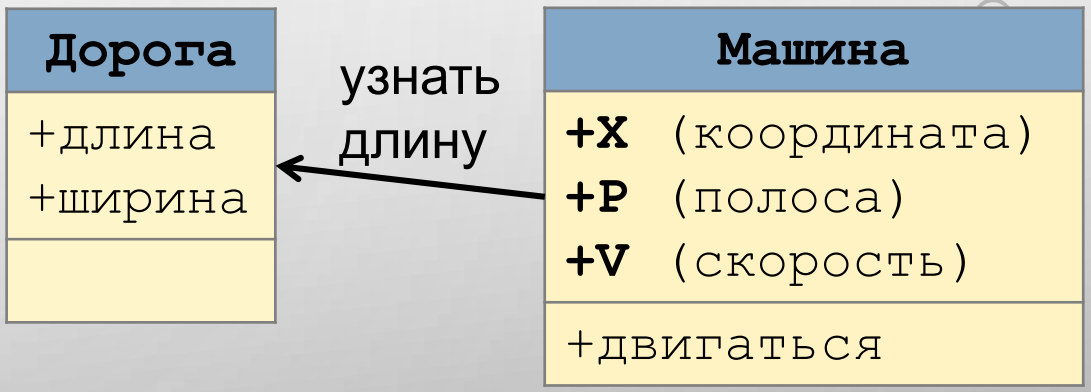
```
11
11
22
33

33

55
```

ООП. Описание классов






```
class TRoad:
  def __init__(self, length0, width0):
    self.length = length0 if length0 > 0 else 0
    self.width = width0 if width0 > 0 else 0
```

```
class TCar:
  def __init__(self, road0, p0, v0):
    self.road = road0
    self.P = p0
    self.V = v0
    self.X = 0

  def move ( self ):
    self.X += self.V
    if self.X > self.road.length:
      self.X = self.X - self.road.length
```

```
road = TRoad(60,3)
N = 3
cars = []
for i in range(N):
  cars.append( TCar(road, i+1, 2*(i+1)) )
for k in range(100):   # 100 шагов
  for i in range(N):   # для каждой машины
    cars[i].move()
print("После 100 шагов:")
for i in range(N):
  print( cars[i].X )
```

ООП. Инкапсуляция и интерфейс

Инкапсуляция – это скрытие внутреннего устройства объекта.

Интерфейс – это внешние свойства и методы объекта, открытые для использования.

+ Защита внутренних данных
+ Проверка входных данных на корректность
+ Изменение устройства с сохранением интерфейса

Вариант 1

```
class TPen:
  def __init__(self):
    self.color = "000000"

pen = TPen()
pen.color = '1188FF'
```

Машина
+color

Вариант 2

```
class TPen:
  def __init__(self):
    self.__color = "000000"
  def getColor(self):
    return self.__color
  def setColor(self, newColor):
    if len(newColor) != 6:
      self.__color = "000000"
    else:
      self.__color = newColor

pen = TPen()
pen.setColor('FFFF00')
print("цвет пера:", pen.getColor())
```

Машина
-color
+getColor +setColor

Вариант 3

```
class TPen:
  def __init__(self):
    self.__color = 0
  def __getColor(self):
    return "{:06x}".format(self.__color)
  def __setColor(self, newColor):
    if len(newColor) != 6:
      self.__color = 0
    else:
      self.__color = int(newColor, 16)
  color = property(__getColor, __setColor)

pen = TPen()
pen.color = "FFFF00"
print("цвет пера:", pen.color )
```

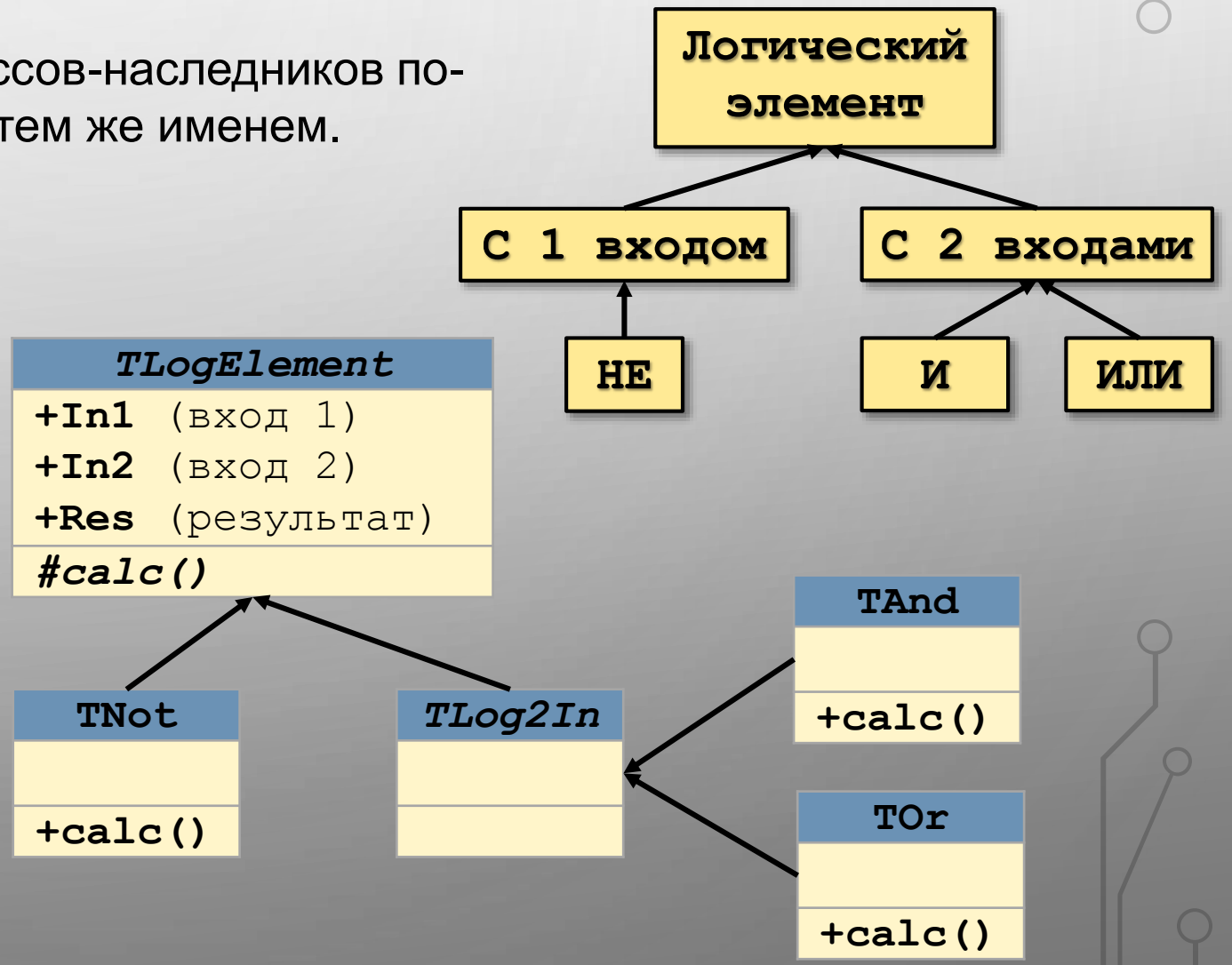
Машина
+color
-getColor -setColor

ООП. Наследование. Пример 1

Абстрактный метод – это метод класса, который объявляется, но не реализуется в классе. Соответственно класс, содержащий хотя бы один такой метод, является **абстрактным классом**.

Полиморфизм – это возможность классов-наследников по-разному реализовать метод с одним и тем же именем.

```
class TLogElement:
  def __init__(self):
    self.__in1 = False
    self.__in2 = False
    self._res = False
    if not hasattr(self, "calc"):
      raise NotImplementedError(\
        "Нельзя создать такой объект!")
  def __setIn1(self, newIn1):
    self.__in1 = newIn1
    self.calc()
  def __setIn2(self, newIn2):
    self.__in2 = newIn2
    self.calc()
  def calc(self):
    pass
  In1 = property(lambda x: x.__in1, __setIn1)
  In2 = property(lambda x: x.__in2, __setIn2)
  Res = property(lambda x: x._res)
```



ООП. Наследование. Пример 1

```
class TLogElement:
  def __init__(self):
    self.__in1 = False
    self.__in2 = False
    self._res = False
    if not hasattr(self, "calc"):
      raise NotImplementedError(\
        "Нельзя создать такой объект!")
  def __setIn1(self, newIn1):
    self.__in1 = newIn1
    self.calc()
  def __setIn2(self, newIn2):
    self.__in2 = newIn2
    self.calc()
  def calc(self):
    pass
  In1 = property(lambda x: x.__in1, __setIn1)
  In2 = property(lambda x: x.__in2, __setIn2)
  Res = property(lambda x: x._res)
```

```
class TLog2In(TLogElement):
  pass
```

```
class TOr(TLog2In):
    def __init__(self):
        TLog2In.__init__(self)
    def calc ( self ):
        self._res = self.In1 or self.In2
```

```
class TAnd(TLog2In):
    def __init__(self):
        TLog2In.__init__(self)
    def calc(self):
        self._res = self.In1 and self.In2
```

```
class TNot(TLogElement):
  def __init__(self):
    TLogElement.__init__(self)
  def calc(self):
    self._res = not self.In1
```

```
elNot = TNot()
elAnd = TAnd()
print("  A | B | not(A&B) ");
print("--------------------");
for A in range(2):
  elAnd.In1 = bool(A)
  for B in range(2):
    elAnd.In2 = bool(B)
    elNot.In1 = elAnd.Res
    print(" ", A, "|", B, "|", int(elNot.Res))
```

ООП. Наследование. Пример 1

```
class TLogElement:
  def __init__(self):
    self.__in1 = False
    self.__in2 = False
    self._res = False
    self.__nextEl = None
    self.__nextIn = 0
    if not hasattr(self, "calc"):
      raise NotImplementedError(\
        "Нельзя создать такой объект!")

  def __setIn1(self, newIn1):
    self.__in1 = newIn1
    self.calc()
    if self.__nextEl:
      if self.__nextIn == 1:
        self.__nextEl.In1 = self._res
      elif __nextIn == 2:
        __nextEl.In2 = self._res
```

```
elNot = TNot()
elAnd = TAnd()
elAnd.link(elNot, 1)
print("  A | B | not(A&B) ");
print("--------------------");
for A in range(2):
  elAnd.In1 = bool(A)
  for B in range(2):
    elAnd.In2 = bool(B)
    print(" ", A, "|", B, "|", int(elNot.Res))
```

ООП. Обход дерева наследования

```
def instancetree(inst):
    print('Tree of', inst.__class__.__name__)
    def classtree(cls, ind):
        print('--|'*ind + cls.__name__)
        for supercls in cls.__bases__:
            classtree(supercls, ind+1)
    classtree(inst.__class__, 1)
```

```
class A: pass
class B(A): pass
class C(A): pass
class D(B,C): pass
class E: pass
class F(D,E): pass
instancetree(D())
```

```
Tree of F
--|F
--|--|D
--|--|--|B
--|--|--|--|A
--|--|--|--|--|object
--|--|--|C
--|--|--|--|A
--|--|--|--|--|object
--|--|E
--|--|--|object
```

__name__ - это атрибут (модуля, класса или функции), содержащий имя своего объекта.

__class__ - это атрибут объекта, содержащий ссылку на класс, к которому принадлежит данный объект.

__bases__ - это атрибут класса, содержащий ссылки на унаследованные классы

Поиск в дереве наследования происходит:

- **Обычно**: экземпляр класса => класс экземпляра => наследуемый класс, и так далее по дереву снизу вверх до самого верхнего супер-класса.
- **При множественном наследовании**: наследуемые классы просматриваются слева на право, согласно списку наследования.
- **При ромбоидальном наследовании**: сначала просматривается уровень дерева, только затем поиск идут выше.

ООП. Специализация наследования

```
class Super: # абстрактный класс, т.к.
  def method(self):
    print('super metod')
  def delegate(self):
    self.action() # метод не определён

class Inheritor(Super):    # наследует
  pass                     # методы «как есть»

class Replacer(Super):     # Замещает
  def method(self):        # метод
    print('replacer method')

class Extended(Super):     # Расширяет
  def method(self):        # метод
    print('start extended metod')
    Super.method(self)
    print('stop extended method')

class Provider(Super):     # Определяет
  def action(self):        # необходимый
    print('provider action') # метод
```

```
inh = Inheritor()
rep = Replacer()
ext = Extended()
prv = Prvider()
inh.method()
rep.method()
ext.method()
prv.delegate()
```

```
super method

replacer method

start extended method
super method
stop extended method

provider action
```

ООП. UML диаграммы классов

Имя класса
список атрибутов [видимость] [имя]: [тип]
список методов: [видимость] [имя]: [переменные]

ВИДИМОСТЬ	
+	Публичный (Public)
-	Приватный (Private)
#	Защищённый (Protected)

Ассоциация – показывает, что объекты одного класса связаны с объектами другого класса таким образом, что можно перемещаться от одного к другому.

Агрегация – это разновидность ассоциации при отношении между целым и его частями. Причём, существование экземпляров классов-коллекций не зависит от времени существования экземпляра класса-контейнера. Если контейнер будет уничтожен, то его содержимое — нет.

Композиция – более строгий вариант агрегации, имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то его содержимое — тоже.

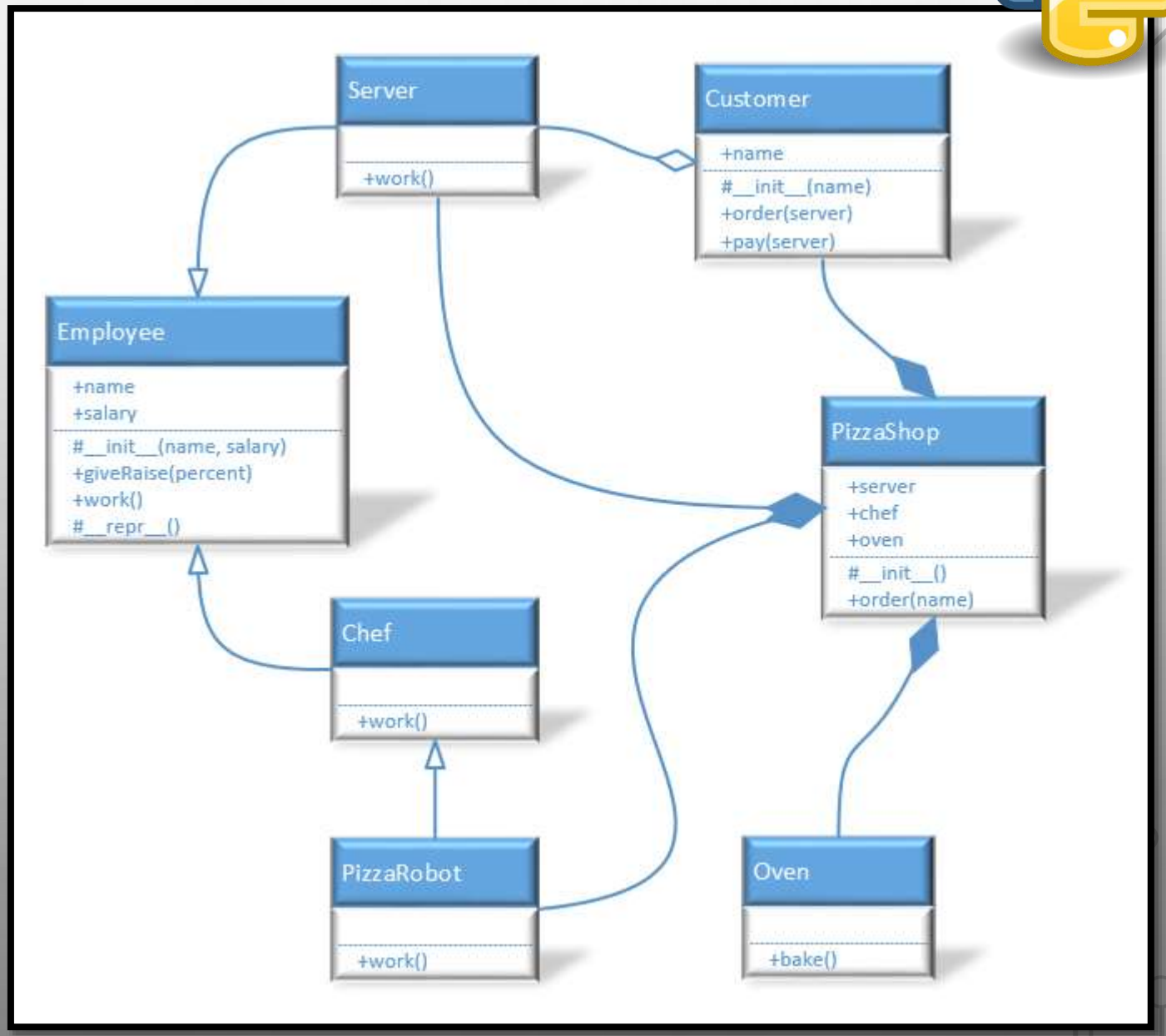
Реализация – отношение между двумя элементами модели, в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком).

Зависимость – отношение между двумя элементами модели, в котором один элемент (клиент) реализует поведение, заданное другим (поставщиком).

ООП. Шаблоны проектирования

Наследование – определяет принадлежность к некому набору методов и атрибутов. Передача атрибутов по дереву наследования сверху-вниз.

Композиция – встраивание других объектов в объект-контейнер, и использование их для реализации методов контейнера.



ООП. Шаблоны проектирования

Делегирование – встраивание объектов в объект-контроллер (класс-обёртка, прокси-класс), который получает запросы на выполнение операций.

```
class Wrapper:
    def __init__(self, obj):
        self.wrapped = obj
    def __getattr__(self, attr):
        print('Trace:', attr)
        return getattr(self.wrapped, attr)
wrList = Wrapper([1,2,3])
wrList.append(5)
```

Фабрика объектов – передача объекта класса в функцию, которая способна динамически управлять процессом создания объектов этих классов

```
def factory(aClass, *args):
   return cClass(*args)
class Spam:
   def doit(self, mes):
      print(mes)
class Person:
   def __init__(self, name, job):
      self.name = name
      self.job = job
obj1 = factory(Spam)
obj2 = factory(Person, 'Vasia', 'chef')
```

ООП. Перегрузка операторов

Перегрузка операторов – это перехватывание встроенных операций с помощью методов класса. Позволяет классам участвовать в обычных операциях, делая их экземпляры похожими на встроенные типы.

Имя метода	Перегружает	Вызывается
`__init__(self, args)`	Создание экземпляра класса	`x = Klass(args)`
`__call__(self, args)`	Вызов объекта класса как функции	`x(args)`
`__iter__(self)`	Вызов итератора объекта	`for el in x: pass; iter(x)`
`__next__(self)`	Вызов следующего значения итератора	`next(x)`
`__str__(self)`	Вызов строкового представления	`print(x); str(x)`

`__getattr__(self, attr)`
`__getattribute__(self, attr)`
`__setattr__(self, attr, value)`
`__delattr__(self, attr)`
	Обращение к атрибуту, присваивание значения или удаление атрибута. Причём:
	`getattr` - к новому атрибуту
	`getattribute`, `setattr`, `delattr` - к любому атрибуту
		`x.new_attr;`
		`x.any_attr = value`
		`del x.any_attr`

`__getitem__(self, key)`
`__setitem__(self, key, value)`
`__delitem__(self, key)`
	Доступ, присваивание или удаление элемента по индексу, либо последовательности по срезу	`x[key]; x[i:j:k]`
		`x[key] = value; x[i:j:k] = value`
		`del x[key]; del x[i:j:k]`

`__len__(self)`	Вызов количества элементов объекта	`len(x)`
`__contains__(self, value)`	Проверка на вхождение элемента в объект	`value in x`

`__lt__(self,other), __gt__(),`
`__le__(), __ge__(),`
`__eq__(), __ne__()`
	Операции сравнения:
	больше, меньше, больше и равно, меньше и равно, равно, не равно.
		`x > y; x < y`
		`x >= y; x <= y`
		`x == y; x != y`

`__add__(self,other), __radd__(),`
`__sub__(), __rsub__(),`
`__mul__(), __rmul__(),`
`__truediv__(), __rtruediv__(),`
`__floordiv__(), __rfloordiv__(),`
`__mod__(), __rmod__()`
	Арифметические операции (лево и право сторонние):
	сложение, вычитание, умножение, деление, целочисленное деление, остаток от деления.
		`x + y; y + x`
		`x - y; y - x`
		`x * y; y * x`
		`x / y; y / x`
		`x // y; y // x`
		`x % y; y % x`

ООП. Декораторы. Описание

Декораторы (**@имя_декоратора**) – это обёртки для других функций, методов, либо классов, которые могут обеспечивать их дополнительным слоем логики.

Простейший декоратор

```
def simpldecor(obj):
  # 1) декорирование
  return obj
                  # функции
@simpldecor       # и методы
def func(*args):
   # 2) действия объекта
```

Декоратор-функция

```
def fdecor(obj):
  # 1) декорирование
  def wrapper(*args):
    # 2) до вызова объекта
    obj(*args)
    # 4) после вызова
  return wrapper

@fdecor  # функции и методы
def func(*args):
   # 3) действия объекта
```

```
@decorator
def func(args): …
```

=

```
def func(args): …
func = decorator(func)
```

Декоратор-класс

```
class Clsdecor:
  # 1) создание декоратора
  def __init__(self,func):
    # 2) декорирование
    self.f = func
  def __call__(self,*args):
    # 3) до вызова
    self.f(*args)
    # 4) после вызова

@Clsdecor  # Только функции
def func(*args):
   # 5) действия объекта
```

Декоратор-класс

```
class ClsDescDecor:
  def __init__(self,func):
    # 1) декорирование метода
    self.f = func
  def __call__(self,*args):
    # 5) вызов экземпляра
    return self.f(*args)
  def __get__(self,inst,own):
    # 3) обращение к методу
    def wrapper(*args):
      # 4) передача аргуметов
      return self(inst,*args)
    return wrapper

class C:     # функции и методы
  def __init__(self): …
    # 2) создание экземпляра
  @ClsDescDecor
  def func(self,*args): …
    # 6) работа метода
```

ООП. Декораторы. Использование.

Декоратор классов

```
def decorCls(cls):
  class Wrapper:
    # 1) декорирование
    def __init__(self,*args):
      # 2) до создания экземпляра
      self.wrp = cls(*args)
      # 4) после создания экз.
    def __getattr__(self,name):
      # 5) обращение к данным экз.
      atr = getattr(self.wrp,name)
      return atr
  return Wrapper

@decorCls      # Только классы
class MyCls: …
  def __init__(self,d): self.d = d
    # 3) во время создания экз.
```

Декораторы часто используются для::

- **Трассировки** функций, методов и классов, т.е. проверки количества их вызовов, времени их выполнения и т.п.
- **Регистрации** объектов в каком-либо интерфейсе.
- **Контроля** за использованием экземпляров класса, с помощью делегирования управления экземпляром классу-обёртке.
- **Проверки значений**, передаваемых аргументам при вызове методов и функций.

Встроенные декораторы

Статические методы (`@staticmethod`) – это методы классов, которые не требуют объект экземпляра `self` для вызова. Вызываются через объект класса, либо экземпляр.

Методы класса (`@classmethod`) – это методы, которым интерпретатор автоматически передаёт класс.

Свойство (`@property`) – позволяет назначить метод, который будет выполняться при обращении (чтении, записи, удалении) к декорированному атрибуту класса.

ООП. Управление атрибутами

Слоты экземпляров – это список имён атрибутов, дающий возможность, ограничить множество возможных атрибутов для экземпляра класса.

- Альтернатива списку-хранилищу атрибутов **__dict__**, который удаляется при использовании **__slots__**.
- Не имеет смысла, если отсутствует у супер-класса, т.к. супер-класс будет иметь словарь **__dict__**.
- Не наследуется в классы-наследники.

```
class Limiter:
   __slots__ = ['age', 'name', 'job']
x = limiter()
print(x.age)       # ошибка: нет значения
x.age = 15
x.weight = 60      # ошибка: недопустимое имя
print(x.__dict__) # ошибка: словарь отсутствует
```

Дескрипторы – это классы, служащие для управления доступом к атрибутам в других классах.

- Дескрипторы позволяют создать свойство «только для чтения», вызвав исключение в методе **__set__()**.
- Дескрипторы могут хранить данные как в собственном пространстве, так и в пространстве экземпляра или класса.
- Если дескриптор имеет отношение только к одному классу, его описание можно вложить в этот класс.

```
class Descriptor:
   'описание'
   def __get__(self, instance, owner): …
   def __set__(self, instance, value): …
   def __delete__(self, instance): …
class Klass:
   attr = Descriptor()
```

ООП. Управление атрибутами

Свойства – это механизм, определяющий методы, вызываемые при обращении к атрибутам экземпляра (на чтение, присваивание, удаление, документирование).

```
attr = property([setter[, getter[, deletter[, doc]]]])
```

Существуют следующие механизмы работы с атрибутами:

1)метод `property`, 2)декоратор `@property`, 3)методы перегрузки класса, 4)дескрипторы

Вариант 1

```
class Person:
    def __init__(self, age, name):
        self.__age = age
        self.name = name
    def __getage(self):
        return str(self.__age)
            + 'years'
    def __older(self, years):
        self.__age += years
    age = property(__getage,
        __older,None)

Lucy = Person(13, 'Lucy')
print(Lucy.age)      # 13 years
Lucy.age = 5         # 18 years
```

Тоже самое с декораторами

```
class Person:
    def __init__(self, age, name):
        self.__age = age
        self.name = name
    @property
    def age(self):
        return str(self.__age)
            + 'years'
    @age.setter
    def age(self, years):
        self.__age += years
```

Вариант 2

```
class Example:
    @property
    def attr(): …
    @attr.setter
    def set_attr(): …
    @attr.deletter
    def del_art(): …
```

ООП. Управление атрибутами

Синтаксис	Описание	Вызов
Методы перегрузки класса		
`__getattr__(self, attr)`	Вызывается при обращении к неизвестному атрибуту (т.е. который отсутствует в экземпляре).	`x.new_attr`
`__setattr__(self, attr, value)`	Вызывается при присваивании значения любому атрибуту. **Примечание.** *Чтобы избежать зацикливание, присваивание следует выполнять через словарь атрибутов*: `self.__dict__['attr'] = value`	`x.any = value`
`__delattr__(self, attr)`	Вызывается при удалении любого атрибута.	`del x.any`
`__getattribute__(self, attr)`	Вызывается при обращении к любому атрибуту. **Примечание.** *Чтобы избежать зацикливание, обращение следует выполнять через метод суперкласса* ***object***: `object.__getattribute__(self, attr)`	`x.any`
Методы класса-дескриптора, который присвоен атрибуту класса-клиента		
`__get__(self, instance, owner)`	Вызывается при обращении к атрибуту экземпляра или класса. • **self** – это ссылка на сам дескриптор, • **instance** – это ссылка на экземпляр класса-клиента, • **owner** – это ссылка на класс-клиент	`x.attr`
`__set__(self, instance, value)`	Вызывается при присваивании атрибуту значения	`x.attr = value`
`__delete__(self, instance)`	Вызывается при удалении атрибута	`del x.attr`
Функции		
`getattr(obj, attr, [default])`	Возвращает значение атрибута **attr** объекта **obj**	
`setattr(obj, attr, value)`	Присваивает значение **value** атрибуту **attr** объекта **obj**	

```
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- ArithmeticError
      |    +-- FloatingPointError
      |    +-- OverflowError
      |    +-- ZeroDivisionError
      +-- AssertionError
      +-- AttributeError
      +-- BufferError
      +-- EOFError
      +-- ImportError
      +-- LookupError
      |    +-- IndexError
      |    +-- KeyError
      +-- MemoryError
      +-- NameError
      +-- OSError
      |    +-- BlockingIOError
      |    +-- ChildProcessError
      |    +-- ConnectionError
      |    +-- FileExistsError
      |    +-- FileNotFoundError
      |    +-- InterruptedError
      |    +-- IsADirectoryError
      |    +-- NotADirectoryError
      |    +-- PermissionError
      |    +-- ProcessLookupError
      |    +-- TimeoutError
      +-- ReferenceError
      +-- RuntimeError
      +-- SyntaxError
      +-- SystemError
      +-- TypeError
      +-- ValueError
      +-- Warning
```

105 Исключения

Обработка исключений

```
try:
   doSomething()
# далее должен выполниться хотя бы один
# из двух операторов: except или finally
except name1:
   print('name1 happened')
except name2, name3:      # не обязательно
   print('name2 or name3 happened')
except Exception as exp: # не обязательно
   print(exp, 'happened')
else:                     # не обязательно
   print('no exception')
finally:                  # не обязательно
   print('this will execute anyway')
```

Собственные исключения

```
class MyException(Exception):
   def __str__(self):
      return 'my exception happened'
```

Возбуждение исключений

инструкция raise

```
raise <класс, либо объект исключения>
```

```
raise Exception('Что-то пошло не так…')
```

инструкция assert

```
assert <условие>, <комментарий>
```

```
assert False, 'исключение возбуждено‘
# возбуждается исключения типа AssertionError
```

инструкция with … as …

```
with <выражение> [as <переменная>]
   <блок with>
```

```
with open(‘file.txt’) as myFile:
# выражение должно поддерживать контекстный
# протокол, файл гарантированно будет закрыт
   for line in myFile:
      print(line)
```

Информация об исключениях

```
sys.exc_info()[0], sys.exc_info()[1] # данная функция возвращает список из 2 элементов
# [0] содержит класс, а [1] содержит экземпляр класса последнего возбуждённого исключения
```

Django

Установка через cmd:

```
pip install Django
```

Создание проекта:

```
django-admin startproject <имя_проекта>
django-admin startapp <имя_проекта>
```

Запуск сайта

```
python manage.py runserver
```

Создание БД:

```
python manage.py makemigrations
python manage.py migrate
```

Создание пользователя:

```
python manage.py createsuperuser
```

Работа с ос

Состав имени файла:

Путь к файлу	+ Имя файла	+ Расширение файла	= Полное имя файла
`'C:\directory\'`	`'filename'`	`'.txt'`	`'C:\directory\filename.txt'`

Права доступа к файлам и директориям:

Например, модификатор mode=0o765 задаёт следующие права:

7 – для владельца, 6 – для группы, в которую входит владелец,

5 – для всех остальных.

Числа в записи прав, означают

Запись	Права на файл	Права на директорию
0	нет	нет
1	выполнение	чтение файлов и их свойств
2	запись	нет
3	запись и выполнение	всё, кроме чтения списка файлов
4	чтение	чтение имён файлов
5	чтение и выполнение	доступ на чтение
6	чтение и запись	чтение имён файлов
7	все права	все права

```
import os,
os.path,
shutil,
pickle,
shelve,
glob
```

Работа с ос.
Функции модуля os

Имя функции	Описание
os.listdir(path)	Возвращает список файлов и поддиректорий в директории **path**
os.mkdir(path[, mode=0o777])	Создаёт директорию с именем **path**
os.makedirs(path[, mode=0o777])	Создаёт директорию с именем **path**, создавая промежуточные директории
os.remove(path)	Удаляет объект файла с именем **path**
os.rmdir(path)	Удаляет пустую директорию **path**
os.removedirs(path)	Удаляет пустую директорию **path**, а также все её пустые родительские директории
os.rename(old, new)	Переименовывает файл или директорию из **old** в **new**
os.renames(old, new)	Переименовывает файл или директорию из **old** в **new**, создавая промежуточные директории
os.access(path, mode)	Проверка доступа к объекту **path** у текущего пользователя, флаг **mode** может принимать значения: **os.F_OK**, **os.R_OK**, **os.W_OK**, **os.X_OK**., т.е. существует, доступ на чтение, запись, исполнение.
os.chmod(path, mode)	Смена прав доступа к объекту **path**, на систему права **mode** – восьмеричное число

Работа с ос.
Функции Модуля os.path

Имя функции	Описание
os.path.exists(path)	True, если путь **path** существует, иначе False
os.path.isfile(path)	True, если путь **path** является файлом, иначе False
os.path.isdir(path)	True, если путь **path** является директорией, иначе False
os.path.isabs(path)	True, если путь **path** является абсолютным, иначе False
os.path.join(path1[, path2[, ...]])	Соединяет пути с учётом особенностей операционной системы
os.path.dirname(path)	Возвращает имя директории пути **path**
os.path.split(path)	Разбивает путь **path** на кортеж (*голова*, *хвост*), где *хвост* - последний компонент пути, а *голова* - всё остальное. *Хвост* никогда не начинается со слеша. Если путь заканчивается слешем, то *хвост* пустой. Если слешей в пути нет, то пустой будет *голова*.
os.path.splitext(path)	Разбивает путь на пару (root, ext), где ext начинается с точки и содержит расширение файла
os.path.splitdrive(path)	Разбивает путь на пару (привод, хвост).
os.path.relpath(path, start=None)	Вычисляет путь к **path** относительно директории **start** (по умолчанию - относительно текущей директории)
os.path.getatime(path)	Возвращает время последнего доступа к файлу **path**, в секундах
os.path.getmtime(path)	Возвращает время последнего изменения файла **path**, в секундах
os.path.getctime(path)	Возвращает время создания файла **path**, в секундах
os.path.getsize(path)	Возвращает размер файла **path**, в байтах

Работа с ос.
Функции модуля shutil

Имя функции	Описание	
shutil.copyfile(src, dst)	Копирует содержимое файла **src** в другой файл **dst**, при этом никакие метаданные не копируются.	Если файл существует, то он будет перезаписан!
shutil.copy(src, dst)	Копирует содержимое файла из исходной пути **src** в путь **dst** назначения вместе с правами доступа.	
shutil.copy2(src, dst)	Копирует содержимое файла **src** из исходной пути в путь назначения **dst** вместе с метаданными.	
shutil.copytree(src, dst)	Рекурсивно копирует целую директорию **src**, в существующую директорию **dst**. Копирование происходит с помощью **shutil.copy2** (по умолчанию).	
shutil.move(src, dst)	Перемещает файл **src** в указанное место **dst** с удалением исходного файла. Копирование происходит с помощью **shutil.copy2** (по умолчанию).	
shutil.rmtree(path)	Удаляет дерево файлов с корнем в **path**, т.е. папку и всё её содержимое.	

Конец