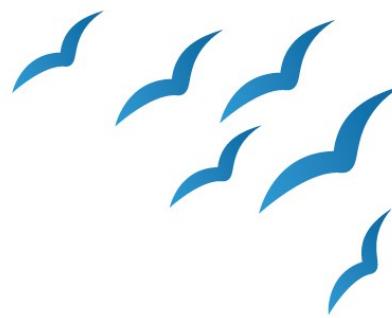




Система контроля версий

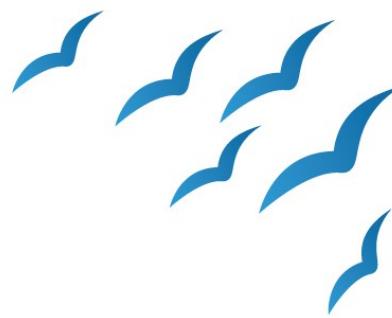




История создания

Проект был создан Линусом Торвальдсом (Linus Benedict Torvalds) для управления разработкой ядра Linux.

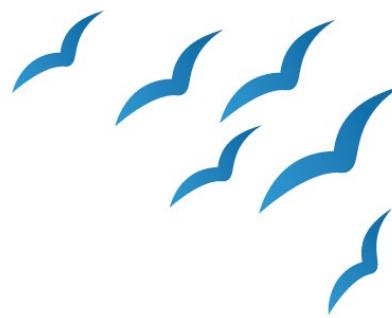
Первая версия выпущена 7 апреля 2005 года меньше чем за неделю разработки.



Цитата

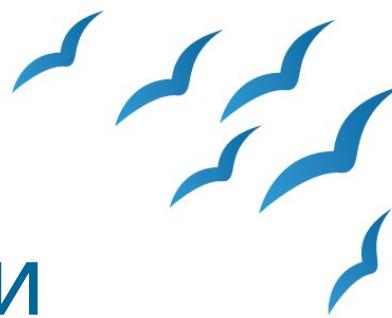
“ I'm an egotistical bastard, so I name all my projects after myself. First Linux, now git. ”

git – (на английском сленге означает мерзавец)



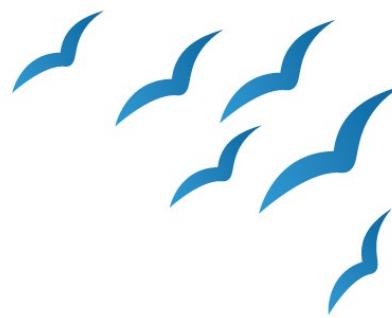
Git – хостинг

- GitHub
- GitLab
- Bitbucket
- SourceForge
- Codebase



Git – система контроля версии

- Летописец
- Машина времени
- Резервная копия
- Мастер параллельных миров
- Народное вече

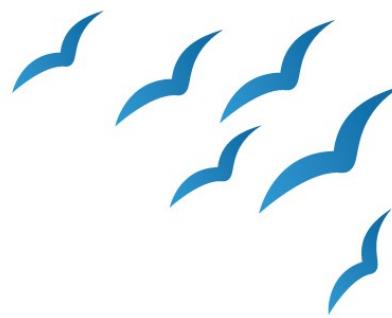


Летописец

Git – ведет всю историю разработки начиная от сотворения проекта.

Делает подшивку черновиков в ветку по контрольным точкам (commit)

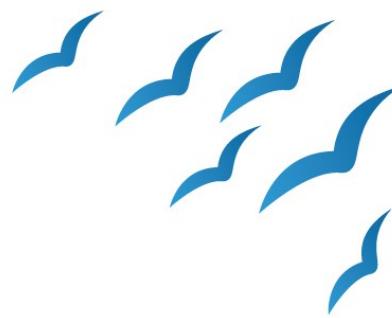
```
> git log
```



Машина времени

Предоставляет возможность путешествовать в прошлое по дереву летописных сводов.

```
>git checkout hash
```



Резервная копия

Создание резервных копий на
удаленных серверах git репозитория.

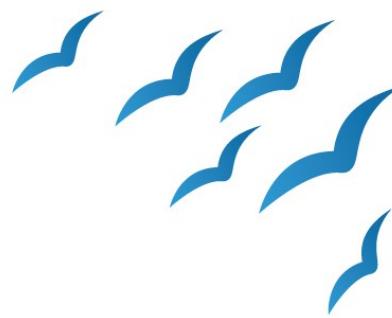
```
>git push origin master
```



Мастер параллельных миров

В Git существует возможность распареллить ветку разработки на несколько , посредством создания дополнительных веток.

```
>git branch new_branch
```



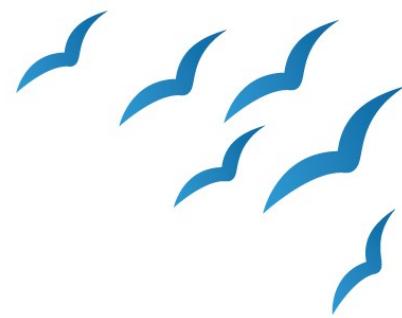
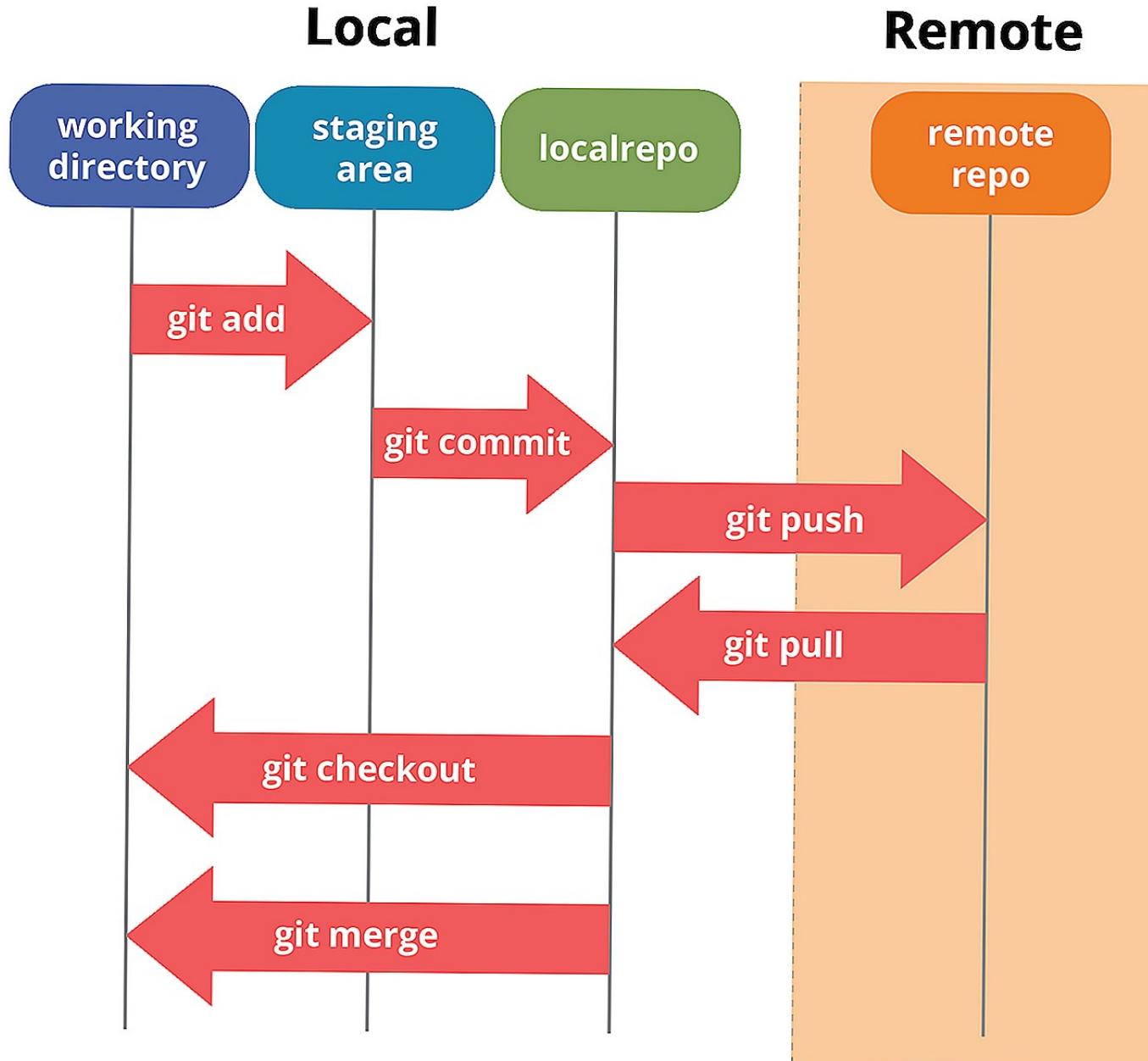
Народное вече

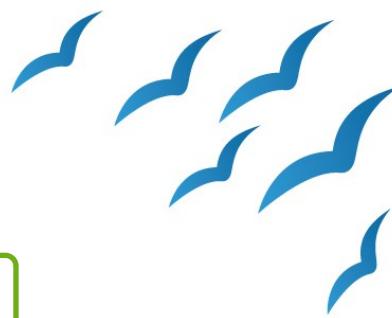
Git воплощает в себе механизмы
коллективного общения разработчиков.
Принятия и обсуждения тех или иных
технических решений. Идея pull request



Управление репозиторием
Git сводится к набору
команд.

Рассмотрим диаграмму.





Настройка

УТВЕРЖДЕНО

2022/05/12 09:24:29

#Устанавливаем в командной строке

```
git config --global user.name "<ваше_имя>"
```

```
git config --global user.email "<адрес_почты@email.com>"
```

Просмотр настроек

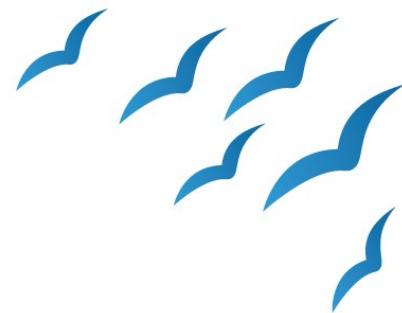
```
>git config --list
```



```
# Превратить каталог, который не находится под  
# версионным контролем, в репозиторий Git
```

>**git init**

```
# После того как была создана папка .git  
# добавляем в текущей каталог файл .gitignore  
# В него заносим все файлы и папки которые не хотим  
# индексировать
```



Индексировать измененный файл

>**git add**

Индексировать измененные файлы

>**git add .**

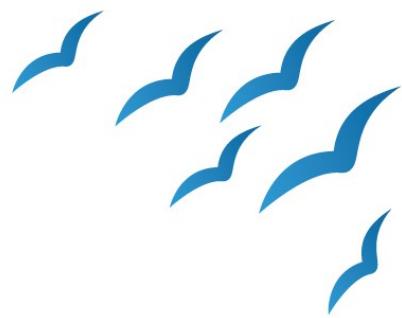


Удаление индексации файла

>**git rm -cashed** имя_файла

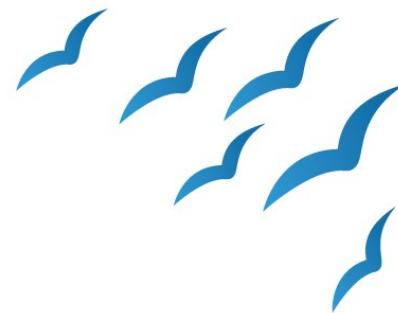
Удаление из индекса и из проекта

>**git rm -f** имя_файла



Просмотр изменений

>**git status**



Фиксация изменения в локальное хранилище
В коммит попадут (будут сохранены) только файлы,
которые были проиндексированы командой git add

>**git commit -m** “комментарии к коду”



Просмотр истории коммитов

>**git log**

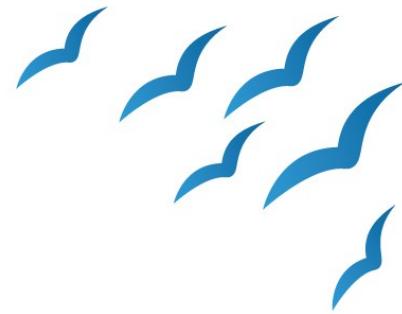
Информация о коммите (метаданные):

1. уникальный идентификатор коммита (хеш);

2. имя и email автора коммита;

3. дата создания коммита;

4. комментарий к коммиту.



Связывание локального репозитория с GitHub

```
>git remote add origin  
git@github.com:my_name/my_repo.git
```

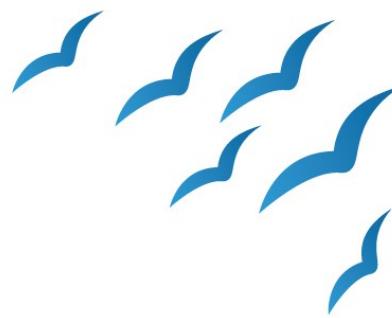
Где my_name – имя пользователя на GitHub
my_repo – название созданного репозитория

Проверка связывания

```
>git remote get-url origin
```

Удаление связывания

```
>git remote remove origin
```



Отправка изменений в удаленный
репозиторий origin ветки master

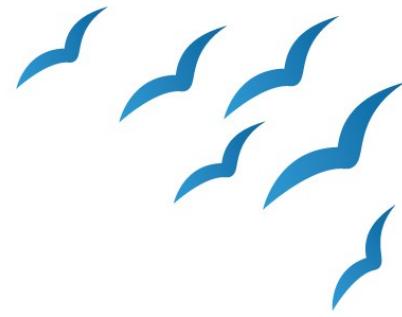
>**git push** origin master

Отправка изменений с текущей ветки

>**git push**

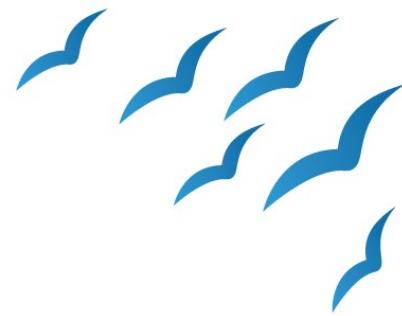
Получение изменений

>**git pull**



Клонировать существующий репозиторий

>**git clone** ссылка-на-репозиторий



Задача !

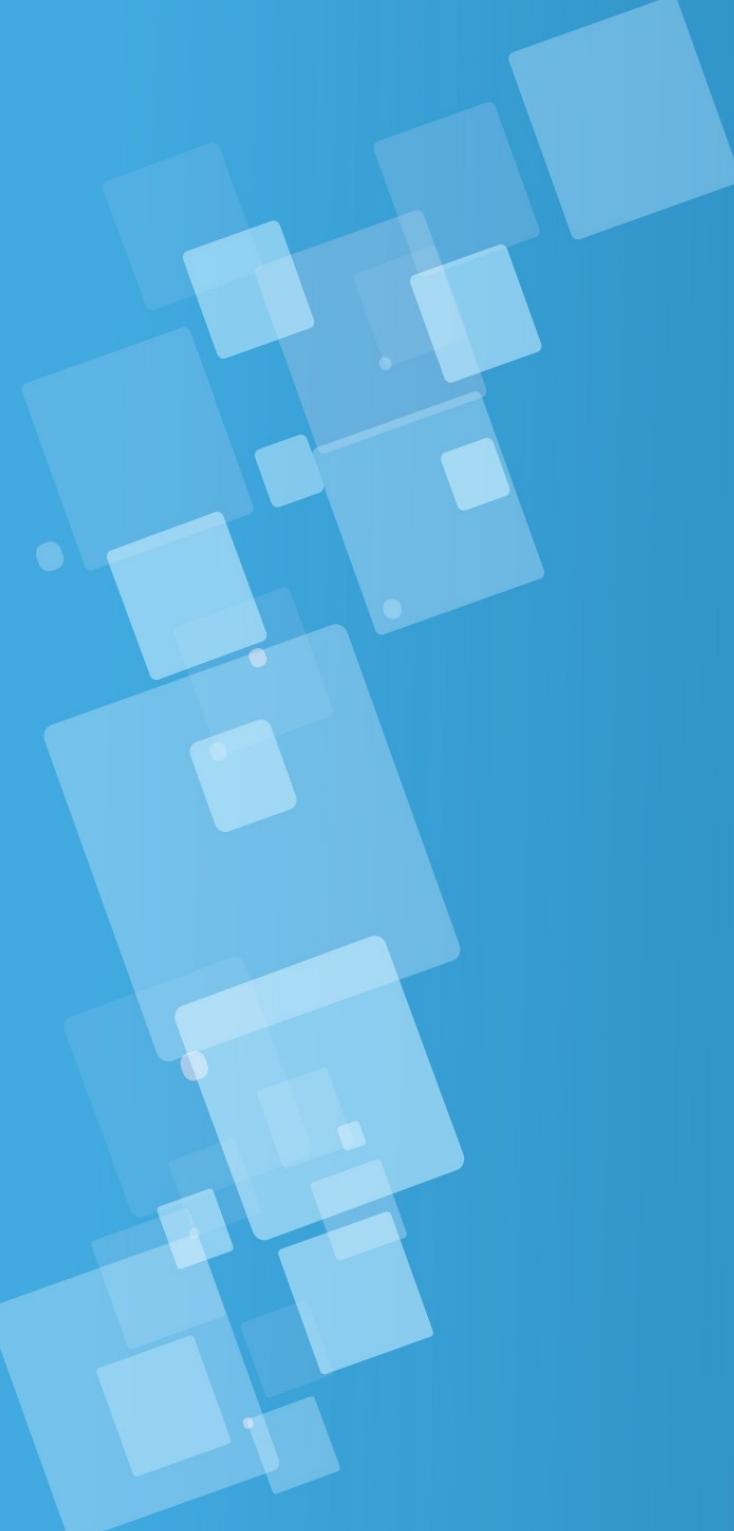
todo: Клонировать существующий репозиторий

```
>git clone  
https://github.com/chertkov-vitaliy/python\_spring\_2022.git
```

Задание



- 1) Создать репозиторий на GitHub с названием
python_spring_work_2022
- 2) Создать папку на локальной машине для домашних работ с названием
python_spring_work_2022
- 3) Превратить созданный каталог в репозиторий Git
- 4) Создать в папке каталога файл README.MD
- 5) Добавить его в локальный репозиторий
- 6) Связать удаленный репозиторий с локальным
- 7) Отправить изменения в удаленный репозиторий



Python

Сфера применения Python

В порядке убывания охвата области и популярности языка в ней:

1. WEB (Django, Flask, aiohttp)
2. Data mining/нейросети (SciPy, NumPy)
3. Тестирование (PyTest)
4. Автоматизация (скрипты)
5. Системные utilы (sys)
6. Desktop-приложения (PyQT)
7. Мобильные приложения (Kivy)

История языка

Язык программирования Python начал свою историю ещё в 1980-х годах, когда идеей о его создании загорелся Гвидо ван Россум - нидерландский программист. В декабре 1989 года он приступил к написанию языка Python в центре математики и информатики в Нидерландах. К 1991 была готова 1 версия интерпретатора.

Особенности языка

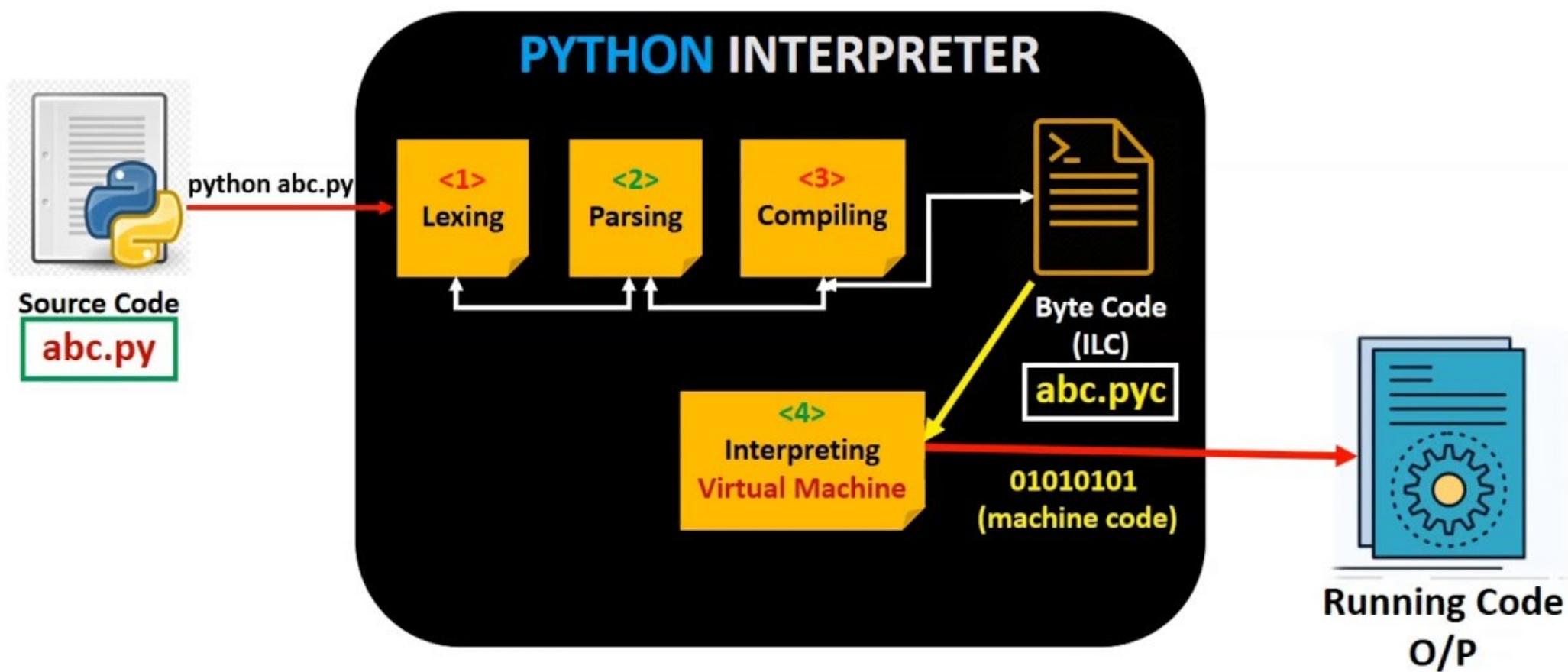
- Интерпретируемый язык высокого уровня
- Динамическая типизация
- Автоматический сборщик мусора
- Поддержка различных парадигм программирования включая объектно-ориентированный подход

Как работает Python ?

1. Программа читается парсером и происходит анализ лексики. Где parser - это анализатор синтаксиса. В итоге получается набор лексем для дальнейшей обработки.
2. Затем парсером из инструкций происходит генерация структуры и формирования дерева синтаксического разбора- AST (Abstract Syntax Tree).
3. После этого компилятор преобразует AST в байт-код и отдает его на выполнение интерпретатору.



Simulating Python Interpreter

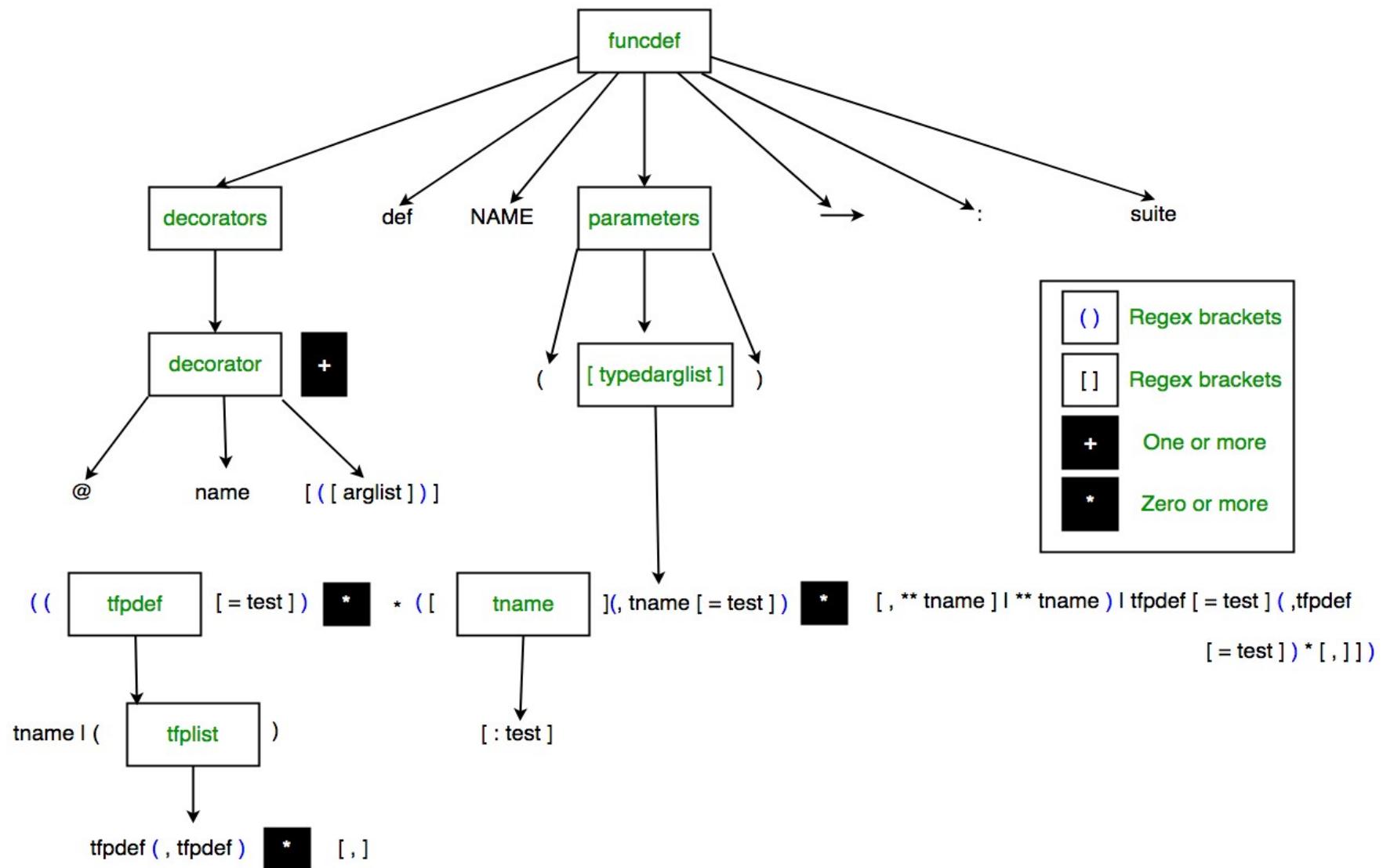




Что будет при интерпретации данного кода ?

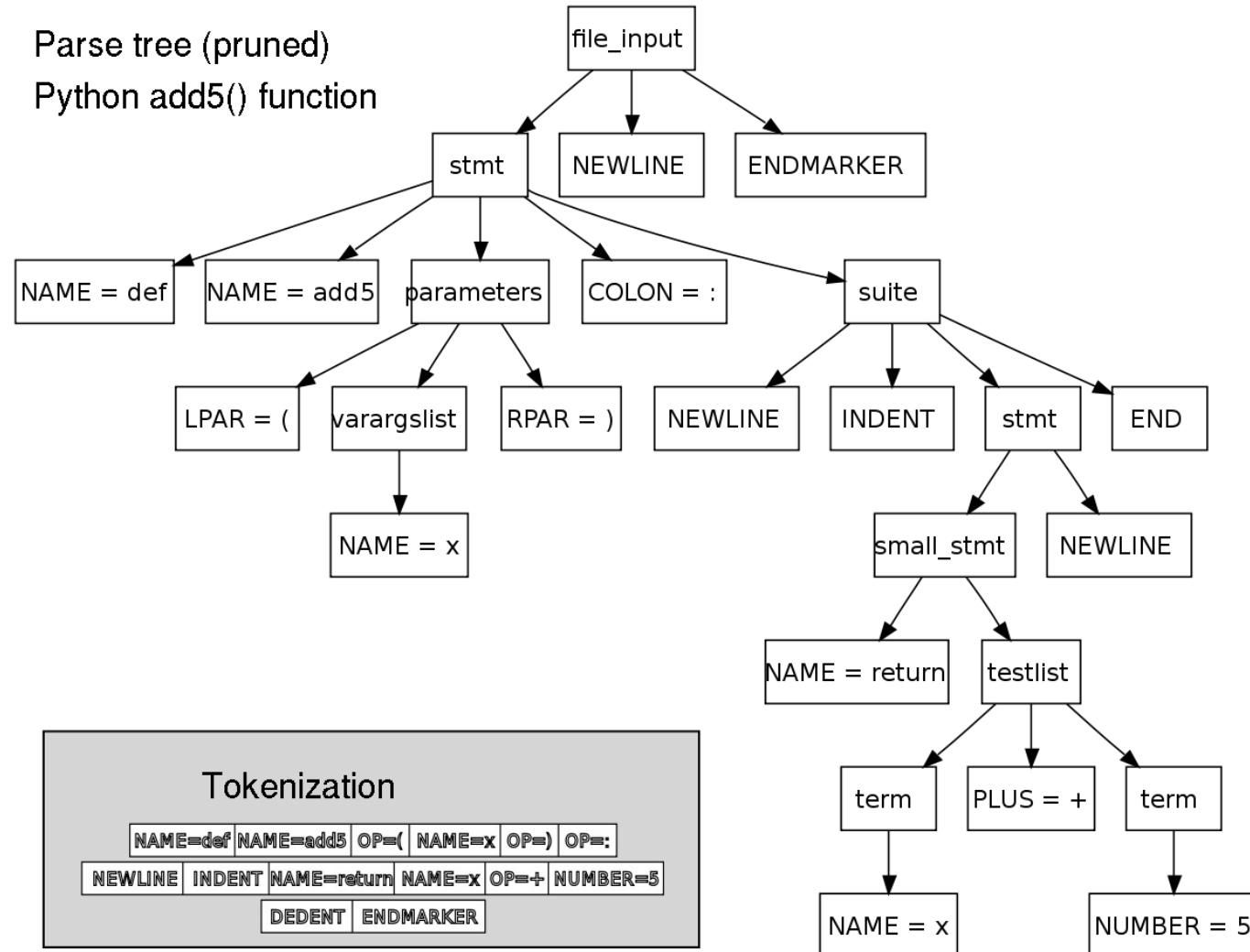
```
def summ5(x):  
    return x + 5
```

Грамматика функции



Дерево синтаксического разбора

Parse tree (pruned)
Python add5() function



Внутренние структуры хранения AST дерева

```
▼ Module: {} 2 keys
  ▼ body: [] 1 item
    ▼ 0: {} 1 key
      ▼ FunctionDef: {} 6 keys
        ► args: {} 1 key
        ► body: [] 1 item
          decorator_list: [] 0 items
          name: "summ5"
          returns: null
          type_comment: null
        type_ignores: [] 0 items
```

На стадии компиляции наш код превращается в байт код. В нашем случае байт код представлен мнемоническими именами. Затем он выполняется на виртуальной машине.

2	0 LOAD_FAST	0 (x)
	2 LOAD_CONST	1 (5)
	4 BINARY_ADD	
	6 RETURN_VALUE	



Для чего нам нужно знать про синтаксический разбор?

Если мы допускаем ошибки в грамматике кода то получаем синтаксическую ошибку.

SyntaxError: invalid syntax



Как писать без ошибок ?

Без ошибок писать пока не получится. Придется их устранять по ходу написания кода.

Чтобы присать без ошибок нужно следовать правилам формальной грамматики языка.

Формальные языки

Любой формальный язык, в том числе и Python, имеет три самые важные составляющие:

- * Операторы
- * Данные
- * Конструкции

Также в языках программирования часто присутствуют комментарии

Рассмотрим как пример один из самых известных формальных языков

(5 * 3 / (1+2))

В данном случае операторами являются

- * Оператор умножения
- * Оператор деления
- * Оператор сложения
- * Операторы группировки (скобочки)

Данными являются

- * Число 5
- * Число 3
- * Число 1
- * Число 2

Основы синтаксиса Python

Программа - это заданная последовательность инструкций.
Инструкции выполняются сверху вниз.

- * Конец строки является концом инструкции (точка с запятой не требуется).
- * Вложенные инструкции объединяются в блоки по величине отступов. Отступ может быть любым. Отступ одинаков в пределах вложенного блока. В Python принят отступ в 4 пробела.

Некоторые операторы языка(if, for, try и т.д) требуют вложенные инструкции. Они в Python записываются в соответствии с одним и тем же шаблоном. Когда основная инструкция завершается двоеточием, за ней идет вложенный блок кода с отступом.

основная инструкция:

вложенный блок

```
if a > b:  
    print(a)
```

4 пробела

Несколько случайных случаев

Иногда возможно записать несколько инструкций в одной строке, разделяя их точкой с запятой:

```
a = 1; b = 2; print(a, b)
```

Но не делайте это слишком часто! Помните об удобочитаемости.
А лучше вообще так не делайте.



Допустимо записывать одну инструкцию в нескольких строках.
Достаточно ее заключить в пару круглых, квадратных или
фигурных скобок:

```
if (a == 1 and b == 2 and
    c == 3 and d == 4): # Не забываем про двоеточие
    print('spam' * 3)
```



Тело составной инструкции может располагаться в той же строке, что и тело основной, если тело составной инструкции не содержит составных инструкций. Пример:

```
if x > y: print(x)
```



Дальнейшее знакомство продолжим
на
практике программирования.

Разберем понятия переменной и типа
переменной.

Задача 1.

Определить в коде переменные:

1. Целочисленного типа
2. Вещественного типа
3. Логического типа
4. Строкового типа
5. Пустого типа

Вывести их типы.

Задача 2.

Преобразуйте переменную age и foo в число

age = "23"

foo = "23abc"

Преобразуйте переменную age в Boolean

age = 123abc

Преобразуйте переменную flag в Boolean

flag = 1

Преобразуйте значение в Boolean

str_one = "Privet"

str_two = ""

Преобразуйте значение 0 и 1 в Boolean

Преобразуйте False в строку.

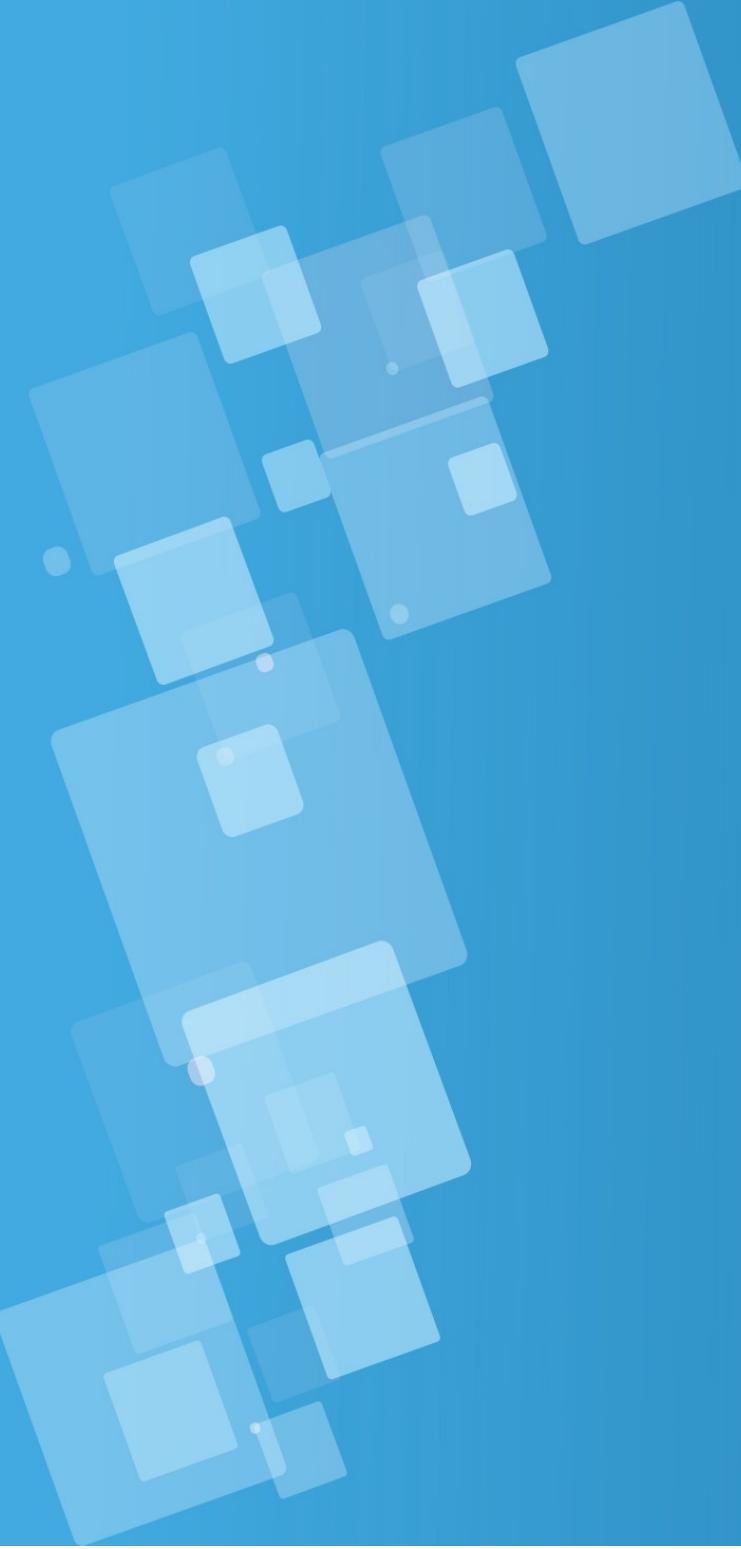
Задача 3.

Данные две переменные:

age = 36.6

temperature = 25

Нужно обменять значения переменных местами. В итого
age должен равняться 25 а temperature - 36.6:



Примитивные типы

Конструкция ветвления

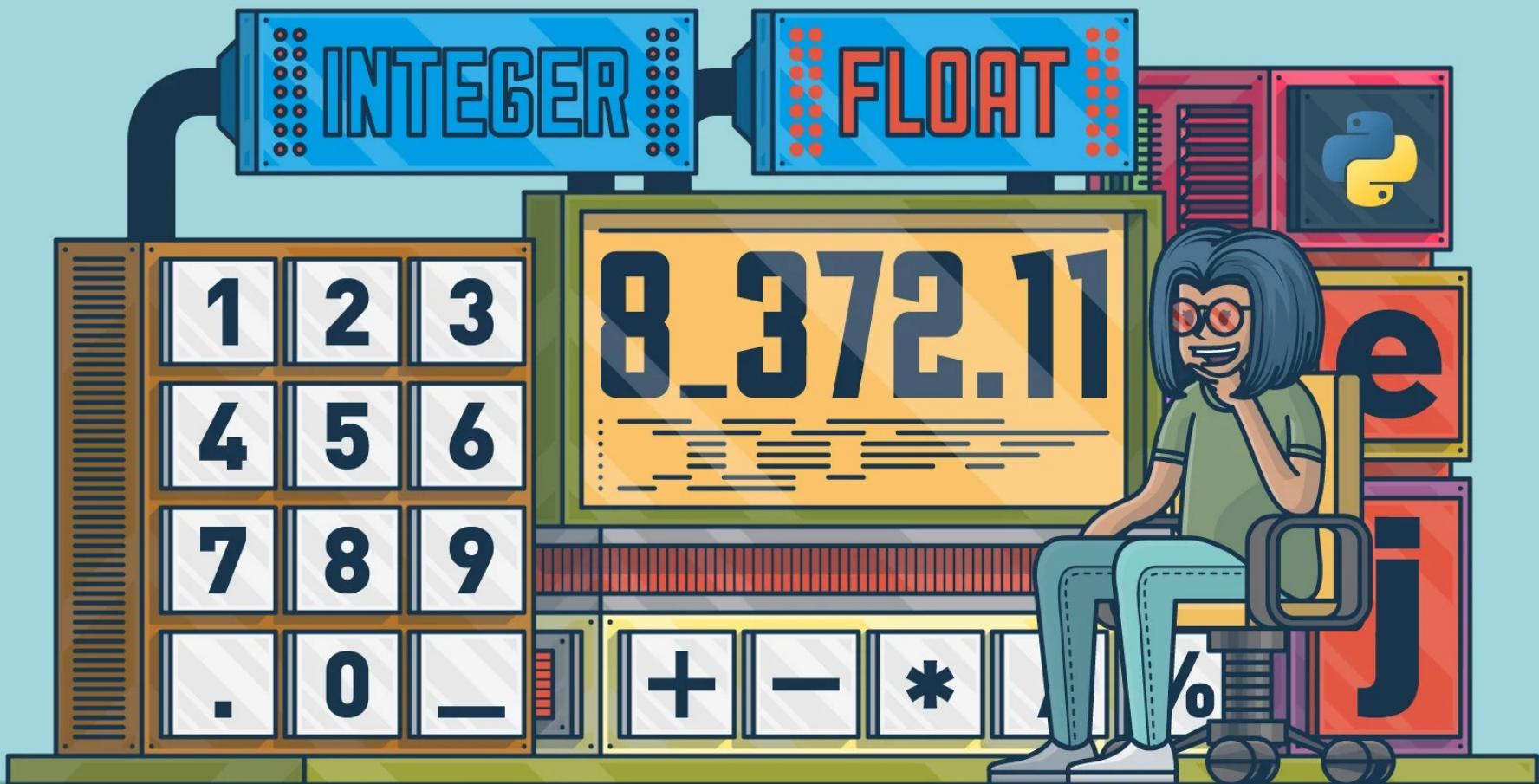
Оглавление

- Примитивные типы
- Таблица истинности
- Конструкции ветвления `if/elif/else`
- Тернарный оператор
- Оператор сопоставления с шаблонами `match/case`
(Python v. 3.10)

Примитивные типы

1. Целые числа (int)
2. Числа с плавающей запятой (float)
3. Комплексные числа (complex)
4. Строки (str)
5. Массивы байт (bytarray)
6. Логический (bool)
7. NoneType

NUMBER



Real Python

Операторы в Python для работы с числами

Операторы в Python для работы с числами:

"+"	(сложение)
"-"	(вычитание)
"*"	(умножение)
"/"	(деление)
"//"	(целочисленное деление)
"%"	(деление с остатком)
"**"	(возвведение в степень)

$\sqrt{25}$ эквивалентно `num ** (0.5)` -?

`abs(x)` - модуль числа

`divmod(x, y)` - пара $(x // y, x \% y)$

`pow(x, y[, z])` x^y по модулю (если модуль задан)

Системы счисления

- Десятичная

```
>>> 7          → int
```

```
>>> 3.14      → float
```

- Двоичная

```
>>> 0b0010    → int
```

- Восьмеричная

```
>>> 0o07      → int
```

- Шестнадцатиричная

```
>>> 0x0F      → int
```

Функции преобразования чисел

- `int(x)` - преобразование к целому числу в десятичной системе счисления.
- `bin(x)` - преобразование целого числа в двоичную строку.
- `hex(x)` - преобразование целого числа в шестнадцатеричную строку.
- `oct(x)` - преобразование целого числа в восьмеричную строку.

Особенности чисел

- Можно работать с большими числами

```
>>x=9999999999999999999999999999999999999999999999999999999999  
9999999999999999999999999999999999999999999999999999999999999999  
9999999999999999999999999999999999999999999999999999999999999999
```

- Форма записи числа

```
>> 9_999_999
```

- Если нужна + бесконечность (inf)

```
>> 2e400 → inf
```

- Если нужна - бесконечность (-inf)

```
>> -2e400 → -inf
```

Scientific notation

- Python использует нотацию Е для отображения больших чисел с плавающей запятой

```
>> 2000000000000000.0      →      2e+17
```

```
>> 1e15      → 1000000000000000.0
```

```
>> 1e16      → ?
```

```
>> 1e-4      → 0.0001
```

```
>> 1e-5      → ?
```

```
>> 1e-0      → ?
```

Юмор из Monty Python

Пример использования Decimal

```
>>> 0.1 + 0.1 + 0.1 - 0.3  
5.551115123125783e-17
```

Для высокой точности следует использовать другие объекты (например decimal и fraction)

```
>>> from decimal import Decimal  
>>> q=w=e=r=t=y=u=i=o=p=Decimal('0.1')  
>>> q+w+e+r+t+y+u+i+o+p  
Decimal('1.0')
```

Комплексные числа

- Комплексное число — это любое число в форме $a + bj$, где a и b — действительные числа, а $j*j = -1$.
- Каждое комплексное число ($a + bj$) имеет действительную часть (a) и мнимую часть (b).

```
>>n = 4 + 3j      → (4+3j)
```

Битовые операторы

~ битовый оператор НЕТ (инверсия, наивысший приоритет);

<<, >> – операторы сдвига влево или сдвига вправо на заданное количество бит;

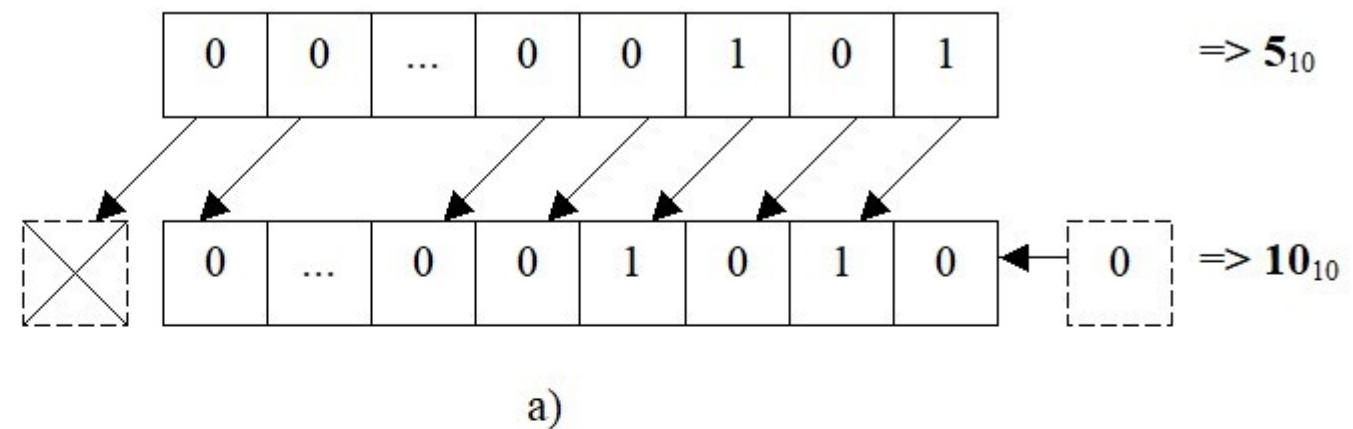
& битовый оператор И

^ битовое исключающее ИЛИ

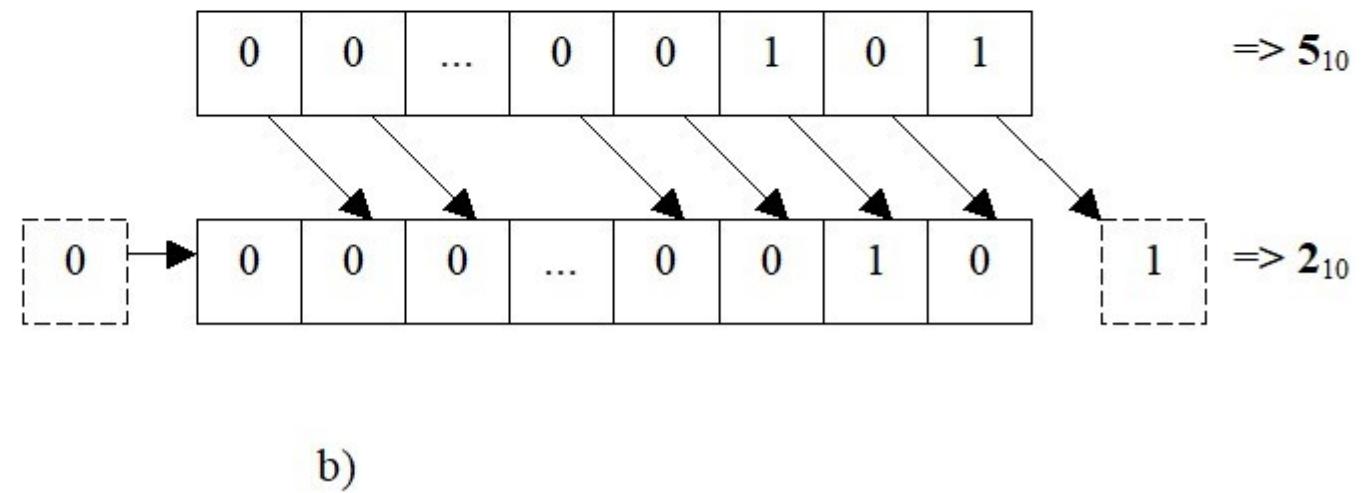
| битовый оператор ИЛИ.

Пример: Операторы сдвига влево `<<`, вправо `>>`

```
x = 5  
y = 5 << 1 # y = 10
```



```
x = 5  
y = x >> 1 # y = 2
```



Операторы сравнения

"=="	(равно)
">="	(больше или равно)
"<="	(меньше или равно)
"!="	(не равно)
"<"	(меньше)
">"	(больше)

Примечание: Когда мы хотим сравнить что две переменные равны то мы делаем так:

```
>> weight_one = 100
>> weight_two = 100
>> weight_one == weight_two      →      true
>> weight_one != 90            →      true
>> weight_one = weight_two      ←      не правильно !!
```

Строки- последовательности символов

Unicode

Строка задается либо парой одинарных " " , либо двойных "" "" или тройных """ """ ковычек. Существенный разницы в Python между одинарными и двойными ковычками нет.

```
>>> name = "" → пустая строка
>>> name = """""" → пустая строка
>>> print(""""
Usage: thingy [OPTIONS]
      -h      Display this usage message
      -H hostname      Hostname to connect to
""")
>>> name = "Peter I"
```

Внимание !!! Частая ошибка.

Не забывайте ковычки при задании строки,
иначе значение будет интерпретироваться
как переменная.

```
>>> str = Hello
```



Максимальная длина строки в Python

Максимальная длина строки зависит от платформы. Обычно это:

- $2^{31} - 1$ — для 32-битной платформы;
- $2^{63} - 1$ — для 64-битной платформы;

Константа maxsize, определенная в модуле sys :

```
>>> import sys >>> sys.maxsize 2_147_483_647
```

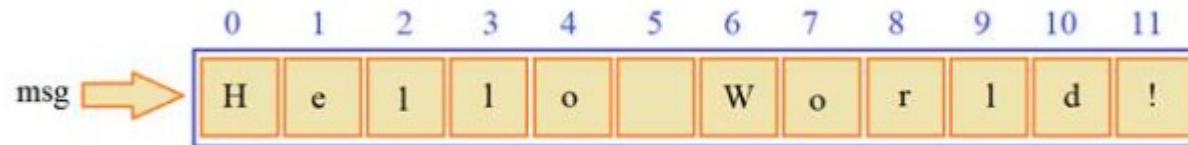
ФУНКЦИЯ len() вычисляет длину строки.

```
>>> len("Hello")      → 5
```

Индексация строки

- Для получения символа в строке нужно обратиться по индексу позиции. Индексация строк начинается с 0

```
>>msg           = "Hello World!"  
>>msg[0]        → "H"
```



Операции сложения и умножения строк

Строки можно складывать (конкатенация строк)

```
>>string = "Hello" + " world !" → "Hello world!"
```

Строки можно умножать на целые числа. Происходит повторение строки n раз.

```
>>> "NO!" * 3 → 'NO!NO!NO!'
```

IMMUTABLE

Строка является неизменяемой (immutable) последовательностью символов.

```
>>msg = "Hello World!"
```

Попытка записать значение в начало слова, вызовет ошибку!

```
>>string[0] = "S"
```

```
TypeError: 'str' object does not support item assignment
```

Строки (str). Срезы (slices)

Срезы (slices) – извлечение из данной строки одного символа или некоторого фрагмента (подстроки)

0	1	2	3	4
H	E	L	L	O
-5	-4	-3	-2	-1

Оператор извлечения среза из строки выглядит так: [X:Y].
X – индекс **начала среза**,
Y – индекс **окончания среза**(символ с номером Y в срезе не входит).

```
>>> s = 'hello'          >>> s = 'hello'  
>>> s[1:4]      ИЛИ    >>> s[-4:-1]  
'ell'           'ell'  
>>>                  >>>
```

Срезы (slice)

#Пустое значение в начале обозначает позицию
0 индекса

```
>>>str = "Hello !"  
>>>str[:3] # Соберем срез по индексам 0, 1, 2  
"He1"  
>>>str[None:3] # Аналогично
```

#Пустое значение в конце обозначает позицию
по концу строки

```
>>str[3:] # Соберем срез по индексам от 2 до 6  
>>"lo !"
```

Проверка вхождения значения в последовательность.

```
>>> "P" in "Python"  
True
```

```
>>> "world" in "Hello world"  
True
```

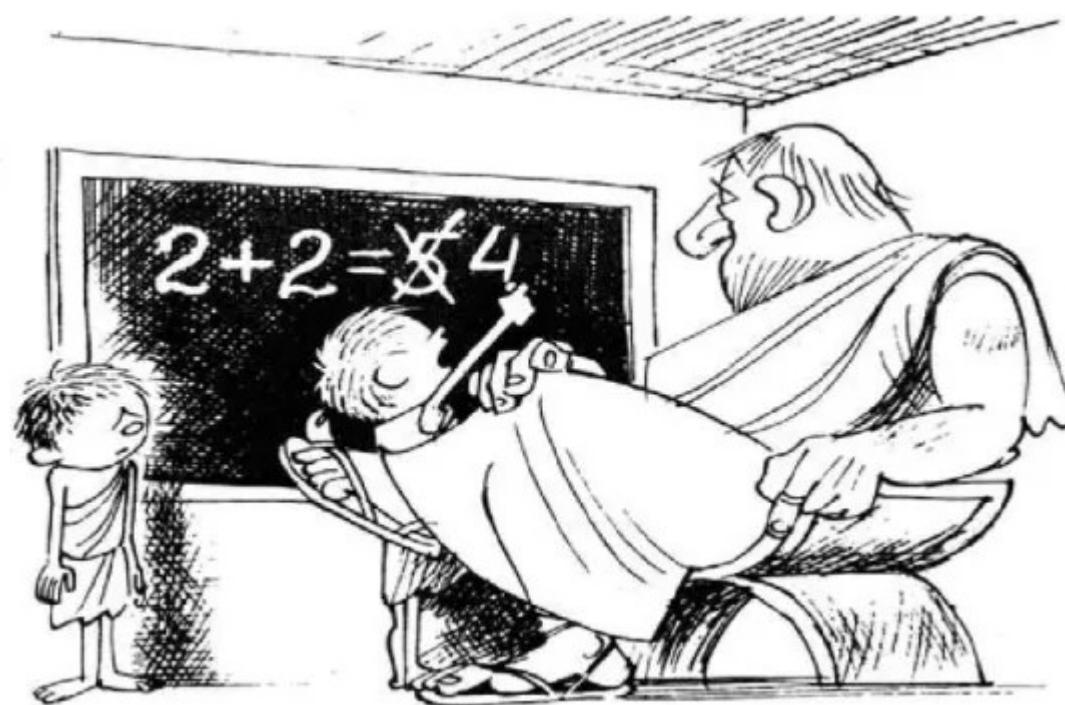
Сравнение строк при помощи == и !=

```
>>>language = 'chinese'  
>>>print(language == 'chinese')      → True  
>>>print(language != 'chinese')     → False
```

```
>>> 'chinese' > 'italiano'
```

Ответ: ?

Питон мне друг но истина дороже.



Логические операторы

AND

— логическое И

OR

— логическое ИЛИ

NOT

— логическое отрицание

IN

— возвращает истину, если элемент присутствует в последовательности, иначе ложь.

NOT IN

— возвращает истину если элемента нет в последовательности.

IS

— проверка идентичности объекта

Таблица истинности

NOT

x	x'
0	1
1	0

AND

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	$x+y$
0	0	0
0	1	1
1	0	1
1	1	1

XOR

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Python - Logical Operators

- not

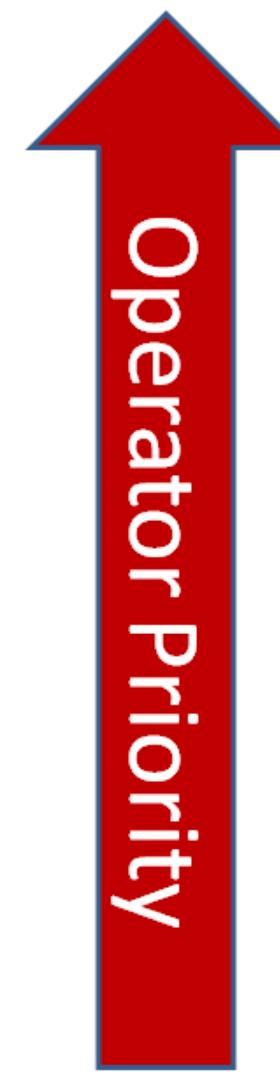
x	not x
False	True
True	False

- and

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

- or

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True



<http://inderpsingh.blogspot.com/>

Применение логических операторов

```
x = 10  
y = 20
```

```
if x > 0 and y > 0:  
    print('Положительные числа')
```

```
if x > 0 or y > 0:  
    print('Хотя бы одно положительное')
```

```
if x > (0 or y) > 0:  
    print('Что будет')
```

Таблица приоритетов операций

Python Operator Precedence

Precedence	Operator Sign	Operator Name
Highest	$**$	Exponentiation
	$+X, -X, \sim X$	Unary positive, unary negative, bitwise negation
	$\ast, /, //, \%$	Multiplication, division, floor, division, modulus
	$+, -$	Addition, subtraction
	$<<, >>$	Left-shift, right-shift
	$\&$	Bitwise AND
	\wedge	Bitwise XOR
	$ $	Bitwise OR
	$==, !=, <, \leq, >, \geq, \text{is}, \text{is not}$	Comparison, Identity
	not	Boolean NOT
	and	Boolean AND
Lowest	or	Boolean OR

Вывод

Чтобы не запутаться в приоритетах операций ставьте в выражении круглые скобки ()

```
# Тестирование порядка выполнения выражения ( слева направо)
```

```
print(4 * 7 % 3)
```

```
# Результат: 1
```

```
print(2 * (10 % 5))
```

```
# Результат: 0
```



ПРИНИМАЙ
РЕШЕНИЕ!

Конструкции ветвления if/else

```
age = input("Enter your age")
age = int(age)
if age <= 18:
    print("Доступ запрещен!")
else:
    print("Доступ разрешен!")
```

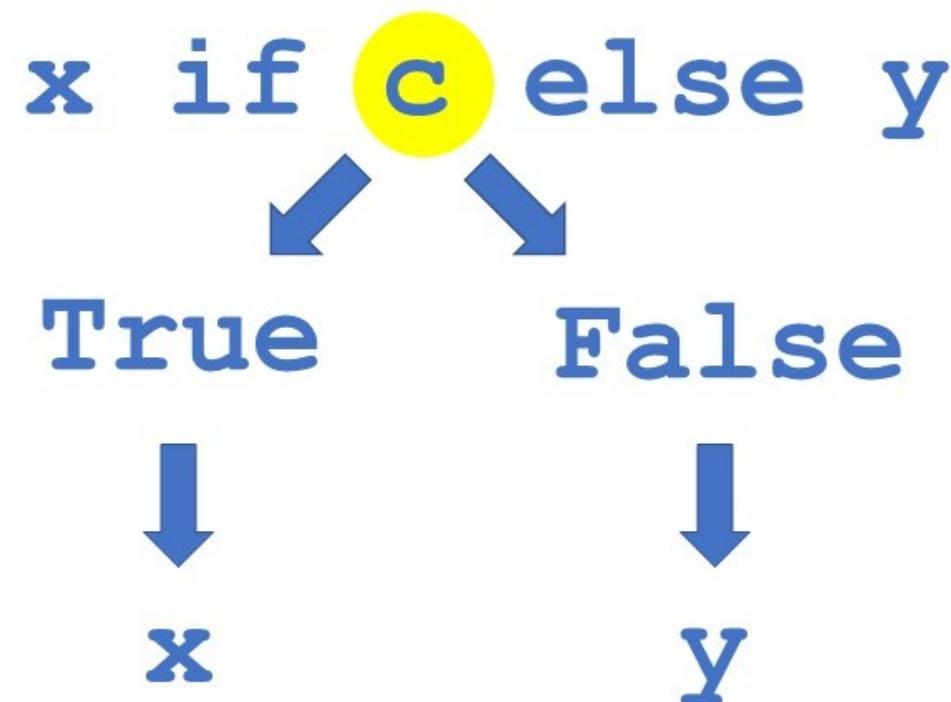
Конструкции ветвления if/elif/else

```
age = input("Enter your age")
age = int(age)
if age <= 16:
    print("Школьник!")
elif 16 < age <= 25:
    print("Студент")
elif 25 < age <= 40:
    print("Сотрудник компании")
else:
    print("Еще не существует!")
```

Конструкции ветвления if

```
str = "Hello"  
if str:  
    print("Не пустая строка!")
```

Ternary Operator



Тернарный оператор

x = 1

y = 2

maximum = x if x > y else y

Pattern Matching in Python 3.10



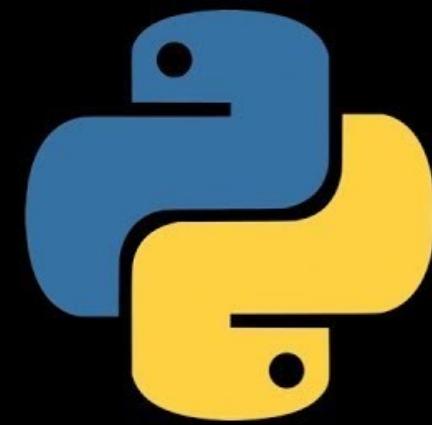
PEP 622

```
match status:  
    case 200:  
        print("OK")  
    case 301 | 302:  
        print("Redirect")  
    case 404:  
        print("Not Found")
```

Пример:

```
color = "RED"

match color:
    case "RED":
        print("Флаг красный")
    case "GREEN":
        print("Трава зеленая")
    case "BLUE":
        print("Небо синее")
```



PYTHON PROGRAMMING

Задача 1

Дано целое число. Если оно является положительным, то прибавить к нему 1; если отрицательным, то вычесть из него 2; если нулевым, то заменить его на 10. Вывести полученное число.

Задача 2.

Задано целое число от 10 до 99 ,
нобходимо найти сумму и произведение
его цифр

Пример:

$$x = 23 \rightarrow 2 + 3$$

Ответ:

сумма - 5

произведение - 6

Задача 3

Задана произвольная строка.
Необходимо разбить ее на две части .
Задачу решить с помощью срезов.

Пример:

str = “Hello world!”

Ответ: Первая часть “Hello”
Вторая часть “world!”

A decorative graphic in the top-left corner consists of several semi-transparent blue squares of varying sizes and shades, arranged in a staggered, overlapping pattern.

Списки

Циклы

Оглавление

- Коллекции
- Конструкция for
- Оператор break
- Оператор continue
- Конструкция while

Коллекции

1. Страна (str)
2. Список (list)
3. Кортеж (tuple)
4. Словарь (dict)
5. Множество (set)
6. Замороженное множество (frozenset)

Список(List)

- Список — это упорядоченный набор элементов, перечисленных через запятую, заключённый в квадратные скобки
- Элементы списка могут быть разных типов, в отличие от элементов массива (array), но, как правило, используются списки из элементов одного типа
- Список может содержать одинаковые элементы, в отличие от множества (set)
- Список можно изменить после создания, в отличие от кортежа (tuple)
- Список может содержать другие списки

Список (mutable)

Создать список можно двумя способами:

Вызывать функцию `list()`

```
lst = list()
```

Использовать квадратные скобки

```
lst = [] → Задали пустой список
```

Пример:

```
lst = list([1, 4, 5])
```

```
lst = list("hello")
```

```
lst = [1, 4, 5]
```

Элементы списка разных типов

Пример:

```
>>> lst = [10, True, [1,2], "#fffffff"]  
>>> type(lst)  
<class 'list'>
```

Список – изменяемый тип данных

Так как список - изменяемый тип данных, то мы можем заменить определённый элемент в нём на другой.

```
>>> mass = [4, 3, 2, 1]
```

```
>>> mass[0] = 8
```

```
>>> [8, 3, 2, 1]
```

Создание копии(клона) списка

```
>>> a = [1, 3, 5, 7]
```

```
>>> b = a[:]
```

Вопрос!

Что за оператор **[:]** ?

Второй способ:

```
>>> a = [1, 3, 5, 7]
```

```
>>> b = list(a)
```

Присвоение списка

В случае, если вы выполните простое присвоение списков друг другу, то переменной **b** будет присвоена ссылка на тот же элемент данных в памяти, на который ссылается **a**

```
>>> a = [1, 3, 5, 7]  
>>> b = a  
>>> b[0] = 10  
>>> print(a)  
[10, 3, 5, 7]
```

Сложение массивов

При сложении происходит объединение множеств массива

```
>>> a = [1, 2]
```

```
>>> b = [3, 4]
```

```
>>> c = a + b
```

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

Размножение списка

Мультипликация списка происходит при умножении его на число.

```
>>> a = [1, 2]
```

```
>>> b*2
```

```
[1, 2, 1, 2]
```

Нахождение значения в списке in , not in

```
>>> a = [1, 2]
```

```
>>> b = 2
```

```
>>> b in a
```

```
True
```

```
>>> 19 not in a
```

```
True
```

```
>>> 1 not in a
```

```
False
```

Получить размер списка - len()

```
>>> mass = [1, 2, 3]  
>>> len(mass)  
3
```

предположение

Догадка, предварительная мысль.



Python List Methods

list •

- append()
- clear()
- copy()
- count()
- extend()
- index()
- insert()
- pop()
- remove()
- reverse()
- sort()



Метод append(x)

Добавление элемента в список осуществляется с помощью метода append()

```
>>> a = []
>>> a.append(3)
>>> a.append("hello")
>>> print(a)
[3, 'hello']
```

Метод clear()

Метод clear() удаляет все элементы из списка.

```
>>> a = [1, 2, 3, 4, 5]
>>> print(a)
[1, 2, 3, 4, 5]
>>> a.clear()
>>> print(a)
[]
```

Метод copy()

Возвращает копию списка. Эквивалентно a[:]

```
>>> a = [1, 7, 9]  
>>> b = a.copy()  
>>> print(b)
```

Метод count (x)

Возвращает количество вхождений элемента **x** в список.

```
>>> a=[1, 2, 2, 3, 3]  
>>> print(a.count(2))  
2
```

Метод `extend(L)`

Расширяет существующий список за счет добавления всех элементов из списка L.

Эквивалентно команде `a[len(a):] = L`

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4]
```

Метод `index(x[, start[, end]])`

Возвращает индекс элемента.

```
>>> a = [1, 2, 3, 4, 5]  
>>> a.index(4)
```

3

Метод `insert(i, x)`

Вставить элемент x в позицию i . Первый аргумент – индекс элемента после которого будет вставлен элемент x .

```
>>> a = [1, 2]
>>> a.insert(0, 5)
>>> print(a)
[5, 1, 2]
>>> a.insert(len(a), 9)
>>> print(a)
[5, 1, 2, 9]
```

Метод `list.pop([i])`

Удаляет элемент из позиции i и возвращает его. Если использовать метод без аргумента, то будет удален последний элемент из списка.

```
>>> a = [1, 2, 3, 4, 5]
>>> print(a.pop(2))
3
>>> print(a.pop())
5
>>> print(a)
[1, 2, 4]
```

Метод `remove(x)`

Удаляет первое вхождение элемента x из списка.

```
>>> a = [1, 2, 3]
>>> a.remove(1)
>>> print(a)
[2, 3]
```

Метод reverse()

Изменяет порядок расположения элементов в списке на обратный.

```
>>> a = [1, 3, 5, 7]  
>>> a.reverse()  
>>> print(a)  
[7, 5, 3, 1]
```

Метод sort()

Сортирует элементы в списке по возрастанию. Для сортировки в обратном порядке используйте флаг reverse=True. Дополнительные возможности открывает параметр key, за более подробной информацией обратитесь к документации.

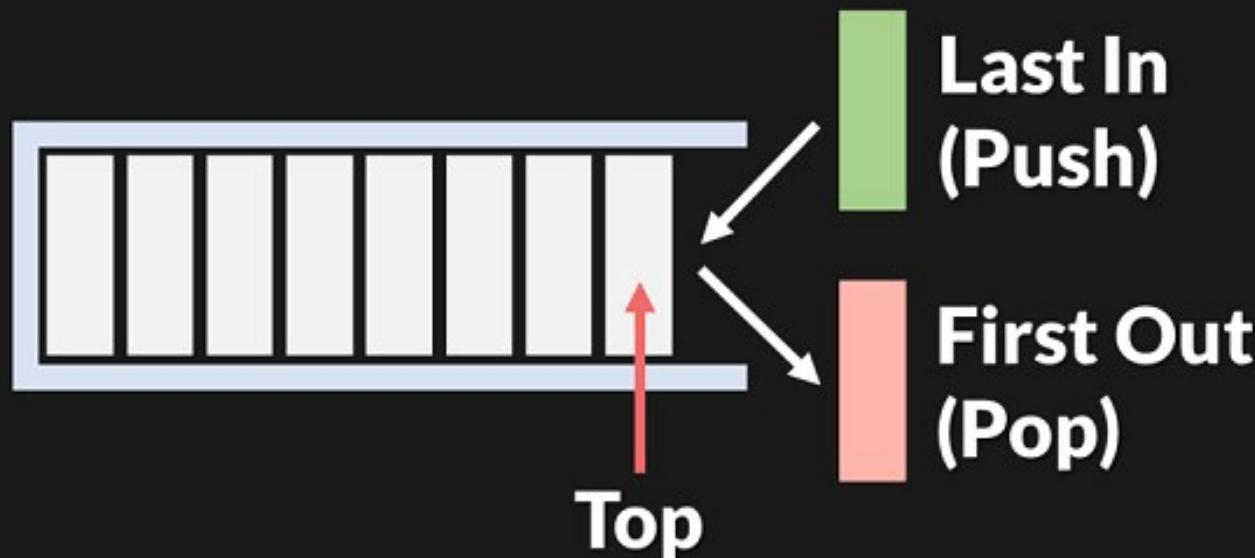
```
>>> a = [1, 4, 2, 8, 1]  
>>> a.sort()  
>>> print(a)  
[1, 1, 2, 4, 8]
```



Как запомнить все эти методы ?



Python Stack Implementation



Срезы(слайсы) в массивах

Срез как и в строках задается тройкой чисел [start:stop:step]
start - индекс с которой нужно начать выборку, stop - конечный
индекс, step - шаг. При этом необходимо помнить, что выборка не
включает элемент определяемый индексом stop.

```
>>> # Получить копию списка
>>> a[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # Получить первые пять элементов списка
>>> a[0:5]
[0, 1, 2, 3, 4]
>>> # Получить элементы с 3-го по 7-ой
>>> a[2:7]
[2, 3, 4, 5, 6]
>>> # Взять из списка элементы с шагом 2
>>> a[::-2]
[0, 2, 4, 6, 8]
>>> # Взять из списка элементы со 2-го по 8-ой с шагом 2
>>> a[1:8:2]
[1, 3, 5, 7]
```

Цикл for

Общая конструкция:

for цель **in** объект:

операторы

```
if проверка: break      # выход из цикла
```

```
if проверка: continue   # переход в начало цикла
```

else:

Операторы # ветка else выполняется если не было выхода с помощью оператора break

Итерация списка с использованием for

```
input_list = [10, "S", 15, "A", 1]  
for x in input_list:  
    print(x)
```

Вывод:

10

"S"

15

"A"

1

Функция range()

Функция `range()` применяется для генерации индексов в цикле `for`. Генерирует диапазон чисел в зависимости от условия.

```
>>> range(5)  
>>> list(range(5))  
[0, 1, 2, 3, 4]  
>>> list(range(-5, 5))  
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
```

Пример:

```
for x in range(3):  
    print('result', x )
```



А можно так ?

```
for x in 3:  
    print('result', x )
```

Оператор break

```
>>> for i in 'hello world':  
...     if i == 'o':  
...         break  
...     print(i * 2, end=' ')  
  
hheelll1
```

Оператор continue

```
>>> for i in 'hello world':  
...     if i == 'o':  
...         continue  
...     print(i * 2, end=' ')  
  
hheelll111 wwwwrrrlldd
```

Волшебное слово else

```
>>> for i in 'hello world':  
...     if i == 'a':  
...         break  
... else:  
...     print('Буквы а в строке нет')  
...  
...  
Буквы а в строке нет
```

Оператор pass

```
for i in 'hello world':
```

```
????           ← Что нужно поставить чтобы for  
                  заработал ?
```

Цикл while

Общая конструкция:

while проверка условия:

операторы

```
if проверка: break      # выход из цикла
```

```
if проверка: continue    # переход в начало цикла
```

else:

Операторы # ветка else выполняется если не было выхода с помощью оператора break

Пример

```
>>> i = 5  
>>> while i < 15:  
...     print(i)  
...     i = i + 2  
...  
...
```

5

7

9

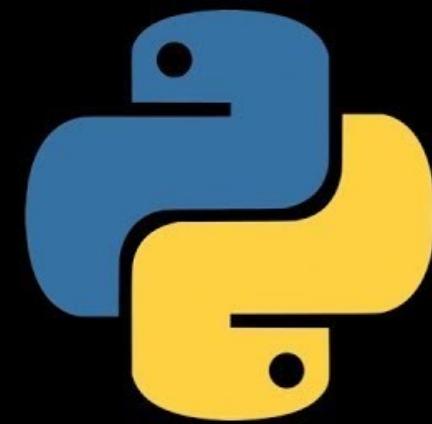
11

13

Бесконечный цикл

```
>>> i = 5  
>>> while True:  
...     print(i)  
...     i = i + 2  
...     if i == 7: break
```

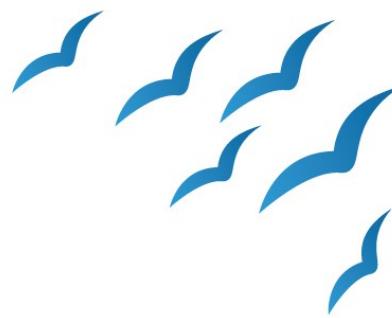
Что выведет код ?



PYTHON PROGRAMMING

Коллекции





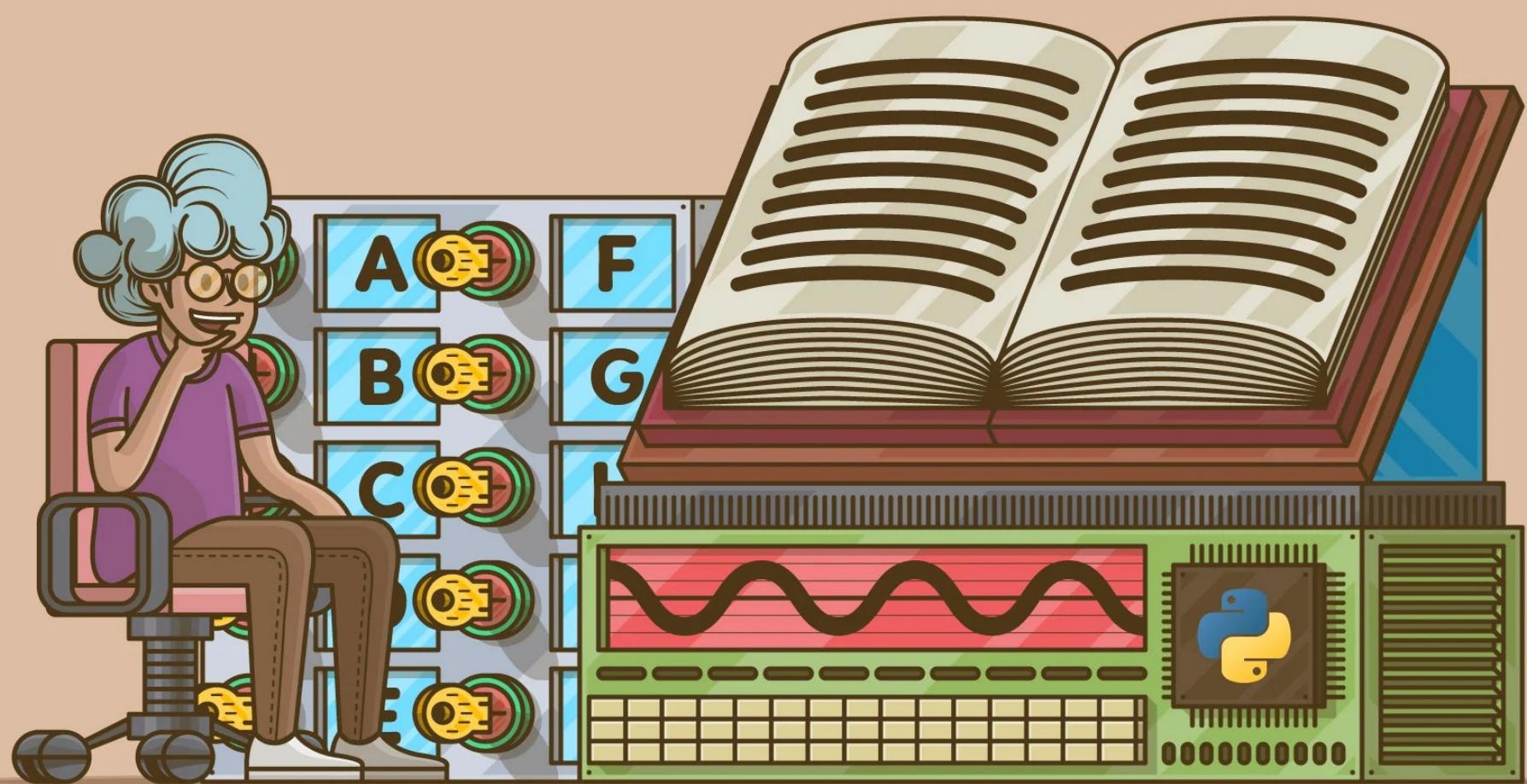
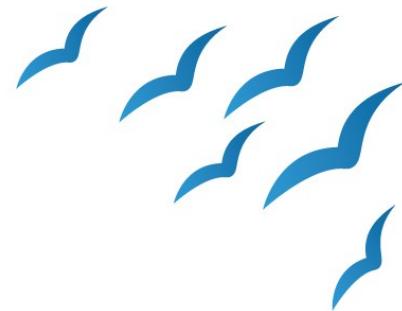
Оглавление

1. Стока (str)
2. Список (list)
3. Словарь (dict)
4. Множество (set)
5. Замороженное множество (frozenset)
6. Кортеж (tuple)

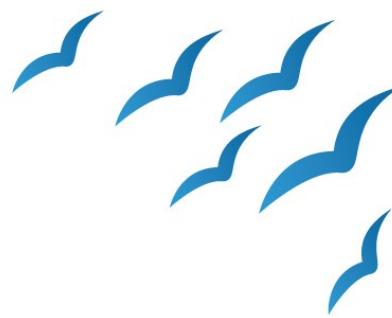


Тип коллекции	Изменяемость	Индексированность	Уникальность	Как создаём
Список (list)	+	+	-	<code>[]</code> <code>list()</code>
Кортеж (tuple)	-	+	-	<code>()</code> , <code>tuple()</code>
Строка (string)	-	+	-	<code>"</code> <code>""</code>
Множество (set)	+	-	+	<code>{elm1, elm2}</code> <code>set()</code>
Неизменное множество (frozenset)	-	-	+	<code>frozenset()</code>
Словарь (dict)	+ элементы - ключи + значения	-	+ элементы + ключи - значения	<code>{}</code> <code>{key: value,}</code> <code>dict()</code>

Словарь



Real Python



Определение словаря

```
dict = {[k]:[v]}
```

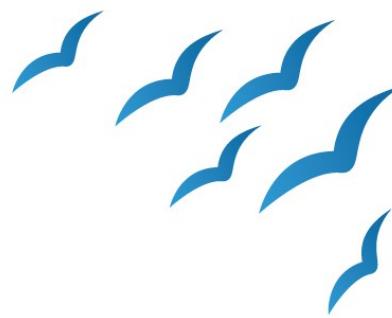
Словарь задается парой **ключ: значение**,

```
dic = {  
    <key>: <value>,  
    <key>: <value>,  
    .  
    .  
    .  
    <key>: <value>  
}
```



Пример 1:

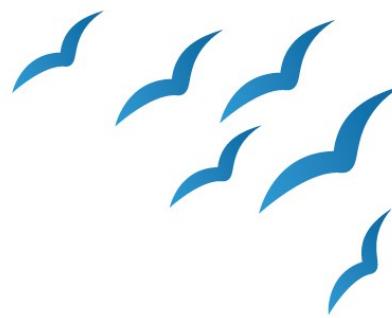
```
person = {  
    'name': 'Маша',  
    'login': 'masha',  
    'age': 25,  
    'email': 'masha@yandex.ru',  
    'password': 'fhei23jj~'  
}  
  
print(type(person))  
<class 'dict'>
```



Пример 2:

#Словарь, где ключи являются целыми числами.

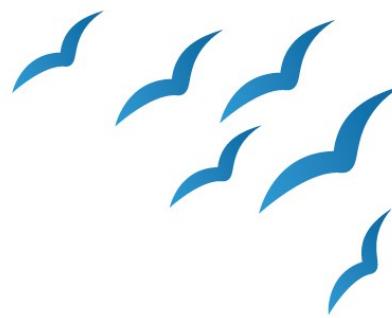
```
dict_sample = {  
    1: 'mango',  
    2: 'coco'  
}
```



Пример 3:

Хмм... если ключи состоят из
примитивных типов то могу ли я
сделать так ?

```
dict_sample = {  
    True: 'mango',  
    False: 'coco'  
}
```



Пример 4:

```
# ... пойдем дальше
```

```
dict_sample = {  
    None: 'mango',  
    None: 'coco'  
}
```

Другие способы создания

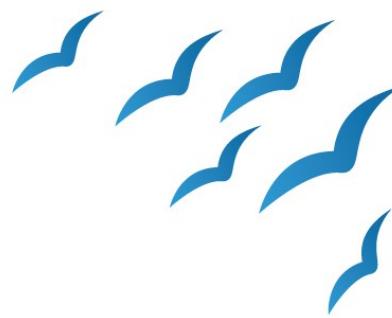


```
{ 'name': 'Маша', 'age': 16 }      # литеральным выражением
```

```
person = {}      # динамическое присваение по ключам  
person['name'] = 'Маша'  
person['age'] = 16
```

!!!

```
dict(name='Маша', age=16) # через конструктор dict
```



Доступ к элементу по ключу

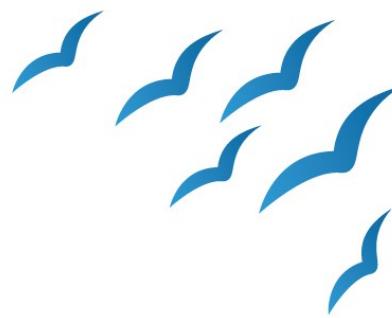
```
dict = { k:v }
```

```
>>> person['name']
```

Маша

```
# Замена значения
```

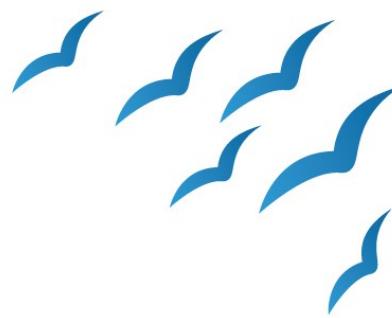
```
>>> person['name'] = 'Даша'
```



Добавление нового элемента

```
dict = { k : v , k2 : v2 }
```

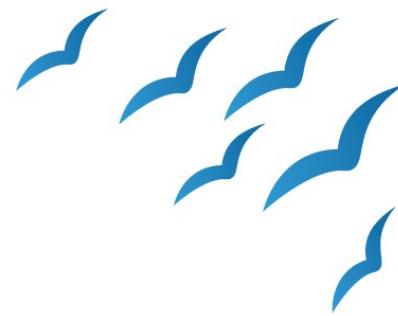
```
>>> person [ 'surname' ] = 'Медведьева'  
{  
    'name': 'Даша',  
    'login': 'masha',  
    'age': 25, 'email': 'masha@yandex.ru',  
    'password': 'fhei23jj~',  
    'surname': 'Медведьева'  
}
```



Удаление элемента

```
dict = { k:v , v:v }
```

```
>>> del person['login']  
{  
    'name': 'Даша',  
    'age': 25,  
    'email': 'masha@yandex.ru',  
    'password': 'fhei23jj~',  
    'surname': 'Медведьева'  
}
```



Проверка на наличие ключа

```
dict = { "A": v }
```

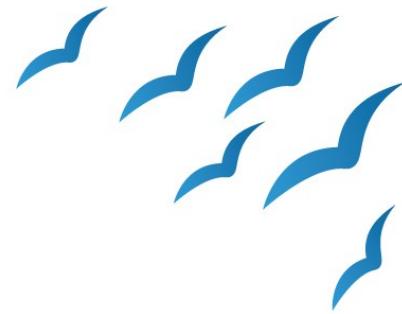
"A"? ↗

```
>>> 'name' in person
```

```
True
```

```
print('Ключ есть') if ('name' in person)  
else print('Ключа нет')
```

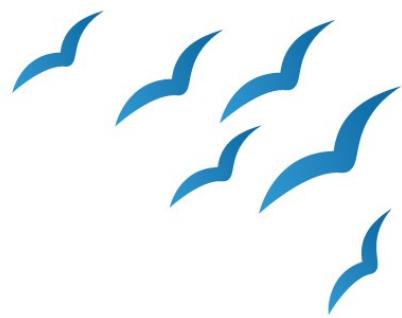
Длина словаря в Python



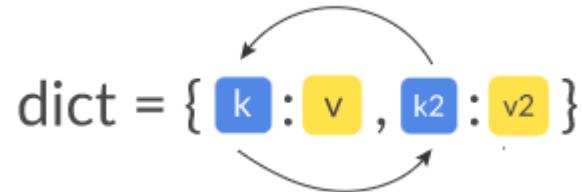
```
dict = { k : v , k2 : v2 }  
       ↓  
len = 2 ↗
```

Количество записей мы можем получить, воспользовавшись функцией `len()`

```
>>> num_of_items = len(person)  
>>> print(num_of_items)  
>>> 5
```



Сортировка словаря



```
statistic_dict = { 'b': 10, 'd': 30, 'e': 15,  
'c': 14, 'a': 33}
```

```
for key in sorted(statistic_dict):  
    print(key)
```

a
b
c
d
e



Итерирование словаря

```
dict = { k : v , k2 : v2 , k3 : v3 }
```



```
statistic_dict = { 'b' : 10, 'd' : 30, 'e' : 15,  
'c' : 14, 'a' : 33}
```

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

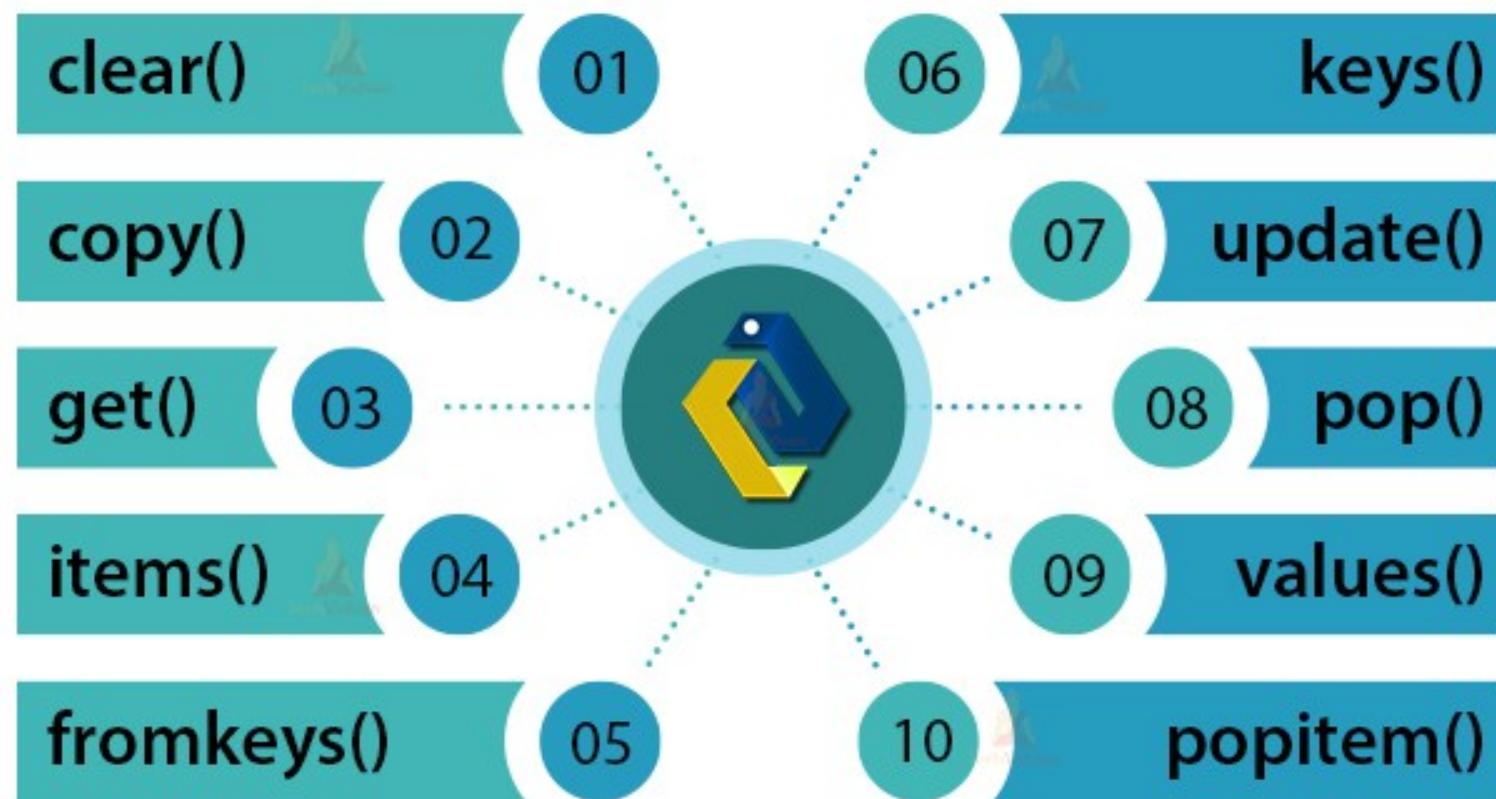
предположение

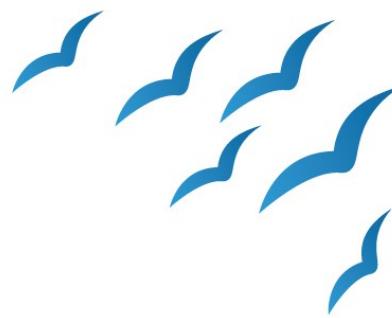
Догадка, предварительная мысль.





Python Dictionary Methods

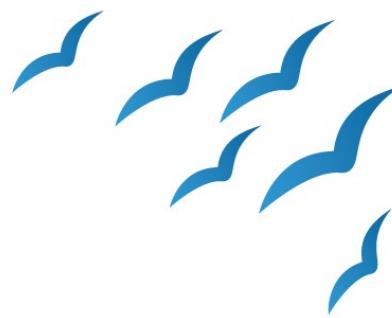




clear()

Метод производит удаление всех элементов из словаря.

```
>>> x = { 'one': 0, 'two': 20, 'three': 3,  
'four': 4}  
>>> x.clear()  
>>> x  
  
# {}
```



copy()

Метод создает копию словаря.

```
>>> x = {'one': 0, 'two': 20, 'three': 3,  
'four': 4}  
>>> y = x.copy()  
  
>>> y  
  
{'one': 0, 'two': 20, 'three': 3, 'four': 4}
```



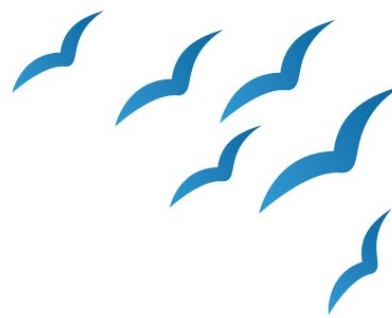
get(key[, default])

Метод dict.get() возвращает значение для ключа key, если ключ находится в словаре, если ключ отсутствует то вернет значение default.

Если значение default не задано и ключ key не найден, то метод вернет значение None.

Метод dict.get() никогда не вызывает исключение KeyError, как это происходит в операции получения значения словаря по ключу [dict[key]].

```
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> x.get('two', 0)
# 2
>>> x.get('ten', 0)
# 0
```



items()

Метод `dict.items()` возвращает новый список кортежей вида `(key, value)`, состоящий из элементов словаря.

```
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}  
>>> items = x.items()  
>>> items  
  
dict_items([('one', 1), ('two', 2), ('three', 3),  
('four', 4)])
```



fromkeys(iterable[, value])

Метод `dict.fromkeys()` встроенного класса `dict()` создает новый словарь с ключами из последовательности `iterable` и значениями, установленными в `value`.

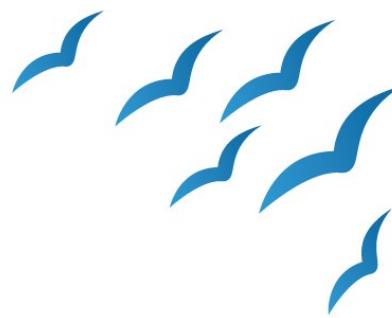
```
>>> x = dict.fromkeys(['one', 'two', 'three',  
'four'])  
  
>>> x  
  
{'one': None, 'two': None, 'three': None, 'four':  
None}  
  
>>> x = dict.fromkeys(['one', 'two', 'three',  
'four'], 0)  
  
>>> x  
  
{'one': 0, 'two': 0, 'three': 0, 'four': 0}
```



keys()

Метод dict.keys() возвращает новый список-представление всех ключей , содержащихся в словаре dict.

```
>>> x = { 'one': 1, 'two': 2, 'three': 3, 'four': 4 }  
>>> keys = x.keys()  
>>> keys  
dict_keys(['one', 'two', 'three', 'four'])
```



keys()

Список-представление ключей `dict_keys`, является динамичным объектом. Это значит, что все изменения, такие как удаление или добавление ключей в словаре сразу отражаются на этом представлении.

```
# Производим операции со словарем 'x', а все
# отражается на списке-представлении `keys`
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> keys = x.keys()
>>> del x['one']
>>> keys
dict_keys(['two', 'three', 'four'])
>>> x
{'two': 2, 'three': 3, 'four': 4}
```

update() - объединение словарей



```
dict1 = { k1 : v1 } )  
dict2 = { k2 : v2 } ) join
```

```
showcase_1 = { 'Apple': 2.7, 'Grape': 3.5,  
'Banana': 4.4 }
```

```
showcase_2 = { 'Orange': 1.9, 'Coconut': 10 }  
showcase_1.update(showcase_2)
```

```
print(showcase_1)
```

```
> { 'Apple': 2.7, 'Grape': 3.5, 'Banana': 4.4,  
'Orange': 1.9, 'Coconut': 10 }
```

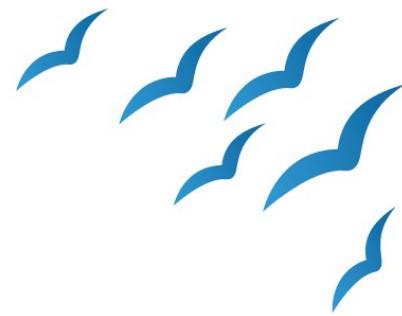
pop(key[, default])



Метод `dict.pop()` вернет значение ключа `key`, а также удалит его из словаря `dict`. Если ключ не найден, то вернет значение по умолчанию `default`.

```
>>> x = { 'one': 0, 'two': 20, 'three': 3}
>>> x.pop('three')
3
>>> x
{ 'one': 0, 'two': 20}
>>> x.pop('three', 150)
150
>>> x.pop('three')
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
#     KeyError: 'ten'
```

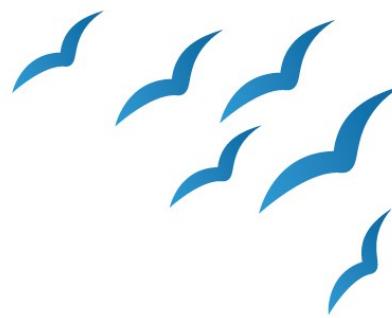
values()



Метод `dict.values()` возвращает новый список-представление всех значений `dict_values`, содержащихся в словаре `dict`.

Список-представление значений `dict_values`, является динамичным объектом. Это значит, что все изменения, такие как удаление, изменение или добавление значений в словаре сразу отражаются на этом представлении.

```
>>> x = {'one': 1, 'two': 2, 'three': 3, 'four': 4}  
>>> values = x.values()  
>>> values  
# dict_values([1, 2, 3, 4])
```



popitem()

Метод `dict.popitem()` удалит и вернет двойной кортеж (`key, value`) из словаря `dict`. Пары возвращаются с конца словаря, в порядке **LIFO** (последним пришёл – первым ушёл)

```
>>> x = {'one': 0, 'two': 20, 'three': 3}
>>> x.popitem()
('four', 4)
>>> x.popitem()
('three', 3)
>>> x.popitem()
('two', 20)
>>> x.popitem()
# Traceback (most recent call last):
#   File "<stdin>", line 1, in <module>
#     KeyError: 'popitem(): dictionary is empty'
```

Множества set



Множество — неупорядоченный набор элементов.
Каждый элемент в множестве уникален (т. е.
повторяющихся элементов нет) и неизменяем.

```
>>> data_scientist =  
set(['Python', 'R', 'SQL', 'Pandas', 'Git'])
```

```
>> data_engineer =  
set(['Python', 'Java', 'Hadoop', 'SQL', 'Git' ])
```



Задание множества

Что будет при дублировании значения ?

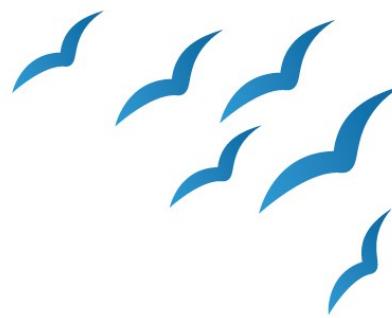
```
>>> data_scientist =  
set(['Python', 'R', 'R', 'SQL', 'Pandas', 'Git'])  
>>> type(data_scientist)  
<class 'set'>
```



Задание множества

Мы также можем создать множество с элементами разных типов. Например:

```
>>> mixed_set = {2.0, "Nicholas", (1, 2, 3)}  
>>> print(mixed_set)  
{'Nicholas', 2.0, (1, 2, 3)}
```



Задание множества

Мы также можем создать множество из списков.

```
>>> num_set = set([1, 2, 3, 4, 5, 6])  
>>> print(num_set)
```

Итерирование множества



```
months = {"Jan", "Feb", "March", "Apr",
          "May", "June", "July", "Aug", "Sep", "Oct",
          "Nov", "Dec"}
```

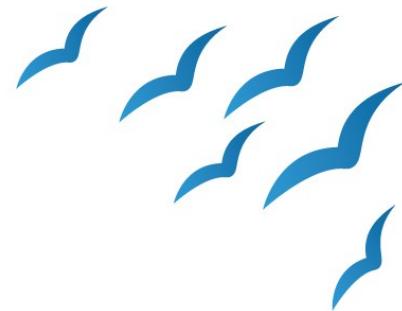
```
for m in months:  
    print(m)
```

```
# проверка на членство в множестве  
print("May" in months)
```

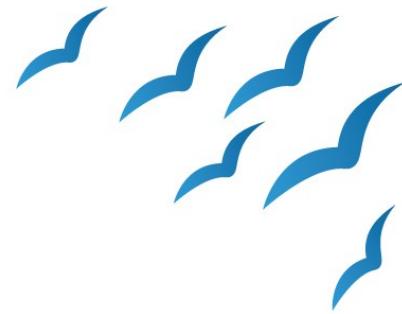
IMPORTANT METHODS IN PYTHON



SET	LIST	DICTIONARY
<ul style="list-style-type: none">• <code>add()</code>• <code>clear()</code>• <code>pop()</code>• <code>union()</code>• <code>issuperset()</code>• <code>issubset()</code>• <code>intersection()</code>• <code>difference()</code>• <code>isdisjoint()</code>• <code>setdiscard()</code>• <code>copy()</code>	<ul style="list-style-type: none">• <code>append()</code>• <code>copy()</code>• <code>count()</code>• <code>insert()</code>• <code>reverse()</code>• <code>remove()</code>• <code>sort()</code>• <code>pop()</code>• <code>extend()</code>• <code>index()</code>• <code>clear()</code>	<ul style="list-style-type: none">• <code>copy()</code>• <code>clear()</code>• <code>fromkeys()</code>• <code>items()</code>• <code>get()</code>• <code>keys()</code>• <code>pop()</code>• <code>values()</code>• <code>update()</code>• <code>setdefault()</code>• <code>popitem()</code>



Добавление элементов

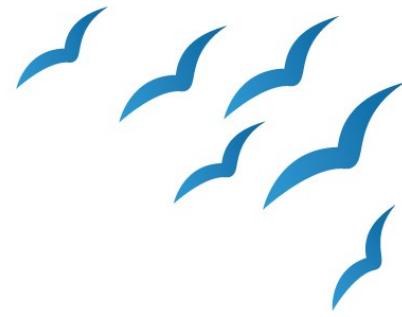


```
months = set(["Jan", "March", "Apr", "May",
"June", "July", "Aug", "Sep", "Oct", "Nov",
"Dec"])
```

```
months.add("Feb")
```

```
print(months)
```

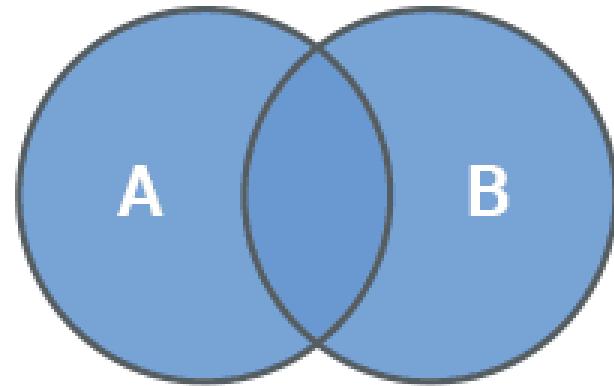
Удаление элемента из множеств



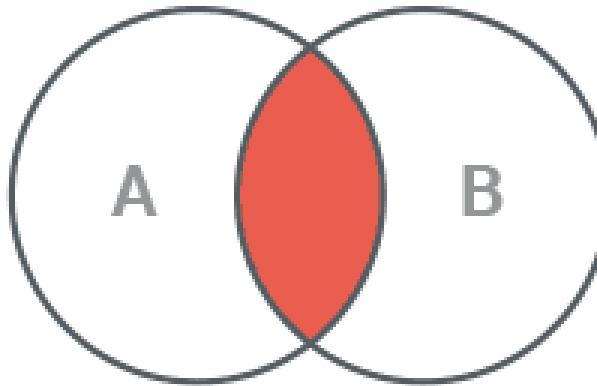
```
>>>num_set = {1, 2, 3, 4, 5, 6}  
>>>num_set.discard(3)  
>>>print(num_set)  
{1, 2, 4, 5, 6}
```

Метод `num_set.remove(7)` аналогичный но вызовет ошибку при отсутствии элемента.

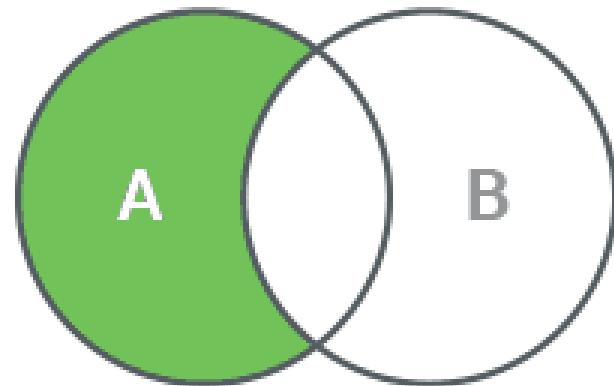
Из теории множеств



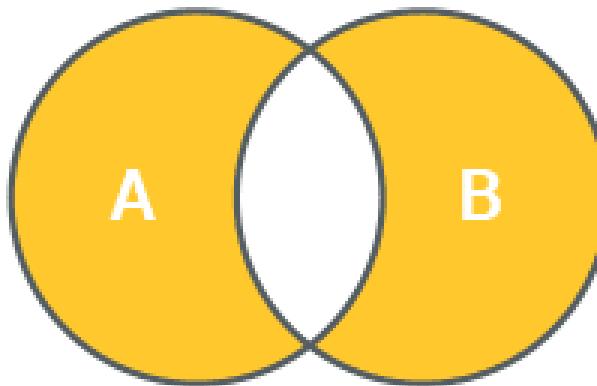
Union



Intersection



Difference



Symmetric Difference

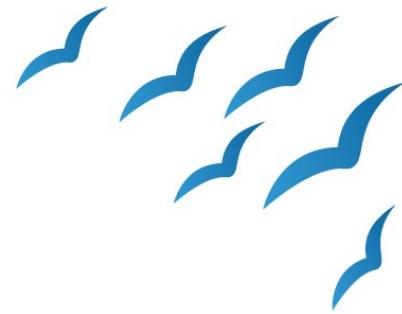


Объединение множеств

```
>>> months_a = set(["Jan", "Feb", "March", "Apr",
"May", "June"])
>>> months_b = set(["July", "Aug", "Sep", "Oct",
"Nov", "Dec"])

>>> all_months = months_a.union(months_b)
print(all_months)
{'Oct', 'Jan', 'Nov', 'May', 'Aug', 'Feb', 'Sep',
'March', 'Apr', 'Dec', 'June', 'July'}
```

union() или оператор |



Объединение может состоять из более чем двух множеств

```
x = {1, 2, 9}
```

```
y = {4, 5, 6}
```

```
z = {7, 8, 9}
```

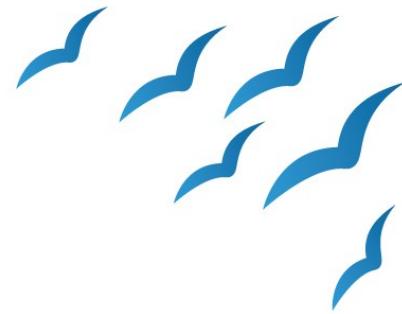
```
output = x.union(y, z)
```

```
print(output)
```

Python

```
{1, 2, 9, 4, 5, 6, 7, 8}
```

```
print(x | y | z )
```



Пересечение множеств

```
x = {1, 2, 3}  
y = {4, 3, 6}  
z = x.intersection(y)  
print(z) #
```

```
x = {1, 2, 3}  
y = {4, 3, 6}
```

```
print(x & y)
```

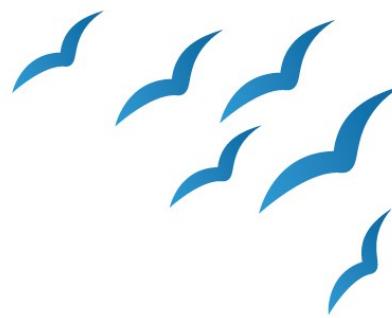
Разница между множествами



```
set_a = {1, 2, 3, 4, 5}  
set_b = {4, 5, 6, 7, 8}
```

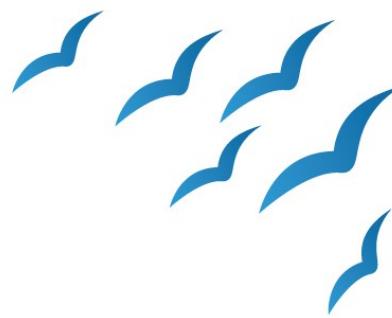
```
diff_set = set_a.difference(set_b)  
print(diff_set)  
{1, 2, 3}
```

```
print(set_a - set_b)
```



Симитричная разница

```
set_a = {1, 2, 3, 4, 5}  
set_b = {4, 5, 6, 7, 8}  
symm_diff = set_a.symmetric_difference(set_b)  
  
print(symm_diff)  
{1, 2, 3, 6, 7, 8}  
  
print(set_a ^ set_b)
```



Сравнение множеств

Чтобы проверить, является ли множество А дочерним от В, мы можем выполнить следующую операцию:

```
months_a = set(["Jan", "Feb", "March", "Apr", "May",  
"June"])
```

```
months_b = set(["Jan", "Feb", "March", "Apr", "May",  
"June", "July", "Aug", "Sep", "Oct", "Nov", "Dec"])
```

```
# Чтобы проверить, является ли множество В  
подмножеством А
```

```
subset_check = months_a.issubset(months_b)
```

```
# Чтобы проверить, является ли множество А  
родительским множеством
```

```
superset_check = months_b.issuperset(months_a)
```

```
print(subset_check)
```

```
print(superset_check)
```



Метод `isdisjoint()`

Этот метод проверяет, является ли множество пересечением или нет. Если множества не содержат общих элементов, метод возвращает `True`, в противном случае — `False`.

```
names_a = {"Nicholas", "Michelle", "John",
"Mercy"}  
names_b = {"Jeff", "Bosco", "Teddy", "Milly"}  
  
x = names_a.isdisjoint(names_b)  
print(x)  
True
```



Frozenset в Python

Frozenset (замороженное множество) – это неизменные множества.

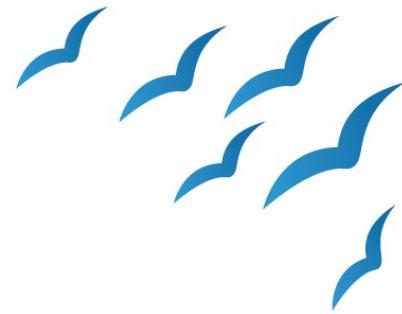
```
X = frozenset([1, 2, 3, 4, 5, 6])
```

```
Y = frozenset([4, 5, 6, 7, 8, 9])
```

```
print(X)
```

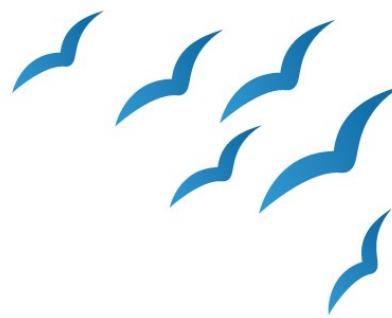
```
print(Y)
```

Какие методы для него не определены?



Карте́жи

- Они являются упорядоченными коллекциями произвольных объектов
- Поддержка доступа по индексу
- Неизменяемые последовательности
- Имеют фиксированную длину
- Представляют из себя массив ссылок на объекты

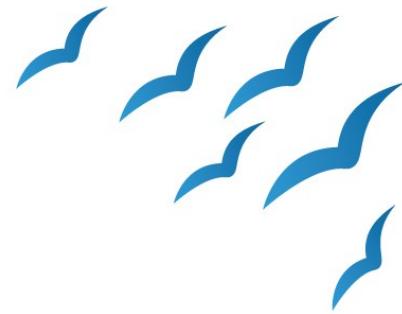


Упаковка картежа

```
# пустой кортеж  
empty_tuple = ()
```

```
# кортеж из 4-х элементов разных типов  
four_el_tuple = (36.6, 'Normal', None, False)  
type(four_el_tuple)  
<class 'tuple'>
```

Упаковка единственного элемента



```
tuple_one = ('a',)
```

```
tuple_two = 'b',
```

```
string_three = 'c'
```

```
print(type(is_tuple))
```

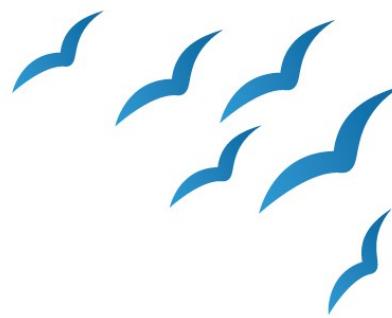
```
print(type(is_tuple_too))
```

```
print(type(string_three))
```

```
<class 'tuple'>
```

```
<class 'tuple'>
```

```
<class 'str'>
```



Природа множественного присваения

```
x, y = 100, 200
```

```
~
```

```
( x, y ) = (100, 200)
```

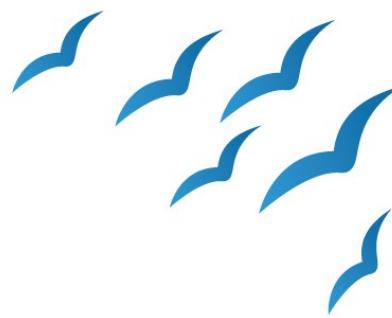
```
print(x)
```

```
100
```

```
print(y)
```

```
200
```

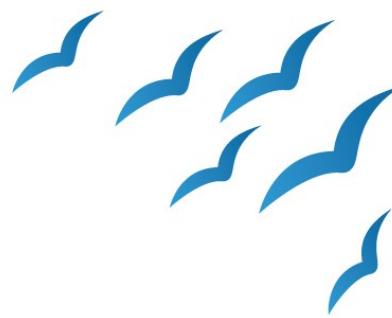
```
x, y = 'ML' - ?
```



Вложенные картежи

```
# пример tuple, что содержит вложенные элементы
```

```
nested_elem_tuple = (('one', 'two'),  
['three', 'four'], {'five': 'six'},  
(('seven', 'eight'), ('nine', 'ten')))  
  
print(nested_elem_tuple)
```



Множественное присвоение(распаковка)

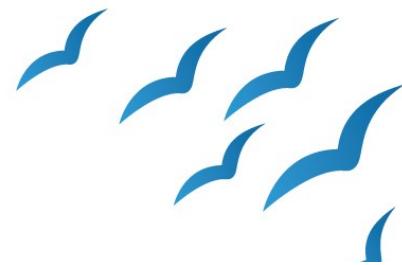
```
notes = ('Do', 'Re', 'Mi', 'Fa', 'Sol', 'La',  
'Si')
```

```
do, re, mi, fa, sol, la, si = notes
```

```
print(mi)
```

```
Mi
```

Индексация



Tuple = ('FACE', 2.0, 97, 'Python', 26.8, 0)



POSITIVE INDEXING

0

1

2

3

4

5

FACE

2.0

97

Python

26.8

0

-6

-5

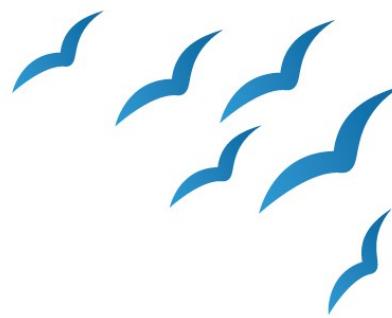
-4

-3

-2

-1

NEGATIVE INDEXING

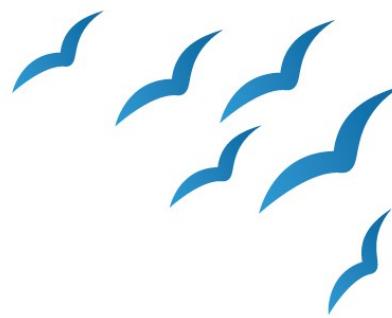


Доступ к элементам

```
>>> a = (1, 2, 3, 4, 5)
>>> print(a[0])
1
>>> print(a[1:3])
(2, 3)
```

Что произойдет ?

```
>>> a[1] = 3
```



Операции

Сложение

(1, 2) + (3, 4)

(1, 2, 3, 4)

Умножение

(1, 2) * 3

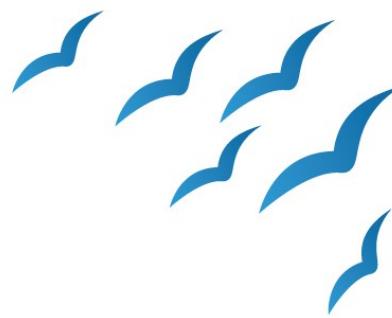
(1, 2, 1, 2, 1, 2)

Вхождение в кортеж

t_str = ('spam',)

'spam' in t_str

True



Сравнение

```
tuple_A = 2 * 2,  
tuple_B = 2 * 2 * 2,  
tuple_C = 'a',  
tuple_D = 'z',
```

```
# при сравнении кортежей, числа сравниваются по  
значению
```

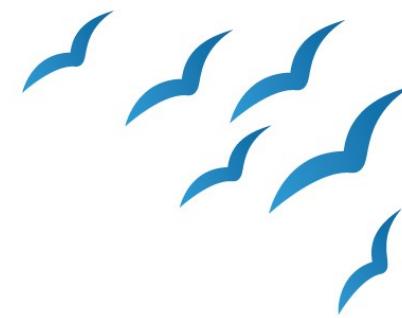
```
print(tuple_A < tuple_B)
```

```
> True
```

```
# строки в лексикографическом порядке
```

```
print(tuple_C < tuple_D)
```

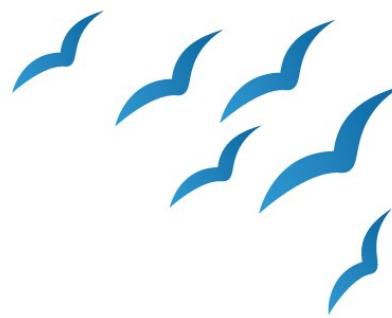
```
> True
```



Итерирование картежа

```
language_token = ('if', 'for', 'while', 'list',
'dict', 'tuple', 'set')
```

```
# Вывести все элементы кортежа
for word in my_tuple:
    print(word)
```



Сортировка

```
not_sorted_tuple = (10**5, 10**2, 10**1, 10**4,  
10**0, 10**3)
```

```
print(not_sorted_tuple)
```

```
> (100000, 100, 10, 10000, 1, 1000)
```

```
sorted_tuple = tuple(sorted(not_sorted_tuple))
```

```
print(sorted_tuple)
```

```
> (1, 10, 100, 1000, 10000, 100000)
```

Удаление



```
some_useless_stuff = ('sad', 'bad things', 'trans  
fats')
```

```
del some_useless_stuff
```

```
print(some_useless_stuff)
```

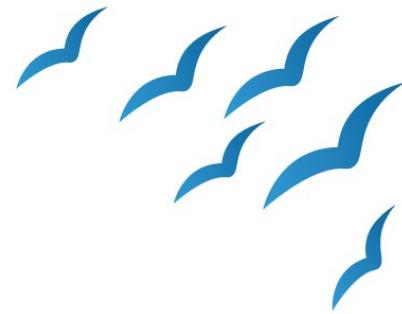
```
>
```

```
Traceback (most recent call last):
```

```
    print(some_useless_stuff)
```

```
NameError: name 'some_useless_stuff' is not  
defined
```

Срезы



Слайсы кортежей **tuple[start:fin:step]**

Где start – начальный элемент среза (включительно), fin – конечный (не включительно) и step – "шаг" среза.

```
float_tuple = (1.1, 0.5, 45.5, 33.33, 9.12, 3.14,  
2.73)
```

```
print(float_tuple[0:3])  
> (1.1, 0.5, 45.5)  
  
# выведем элементы с шагом 2
```

```
print(float_tuple[-7::2])
```

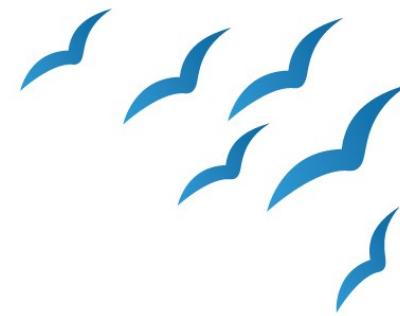
```
> (1.1, 45.5, 9.12, 2.73)
```

Python Tuple Methods

tuple • → count()
 → index()



Индекс заданного элемента **index(value, start, stop)**



```
rom = ('I', 'II', 'III', 'IV', 'V', 'VI', 'VII',  
'VIII', 'IX', 'X')
```

```
print(rom.index('X'))
```

```
9
```

```
str = ('aa', 'bb', 'aa', 'cc')
```

```
print(str.index('aa'))
```

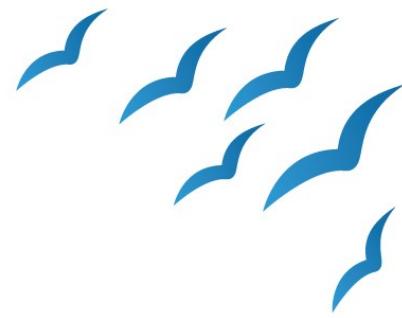
```
0
```

```
str = ('aa', 'bb', 'aa', 'cc' )
```

```
print(str.index('aa', 1, len(str)))
```

```
print(str.index('aa', 1, ))
```

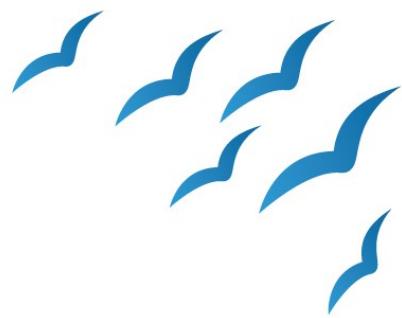
```
2
```



Число вхождений элемента **count()**

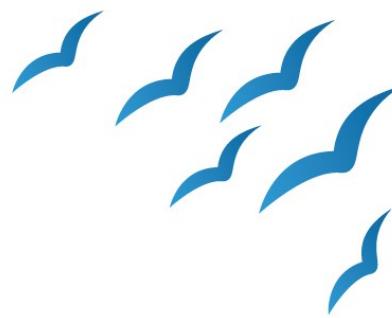
```
t_str = ('aa', 'bb', 'aa', 'cc')  
print(t_str.count('aa'))
```

2



Длина кортежа **len()**

```
python_ = ('p', 'y', 't', 'h', 'o', 'n')  
print(len(python_))
```



Преобразование tuple → str

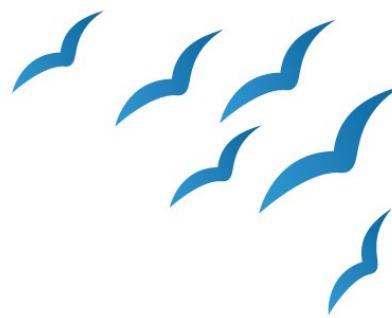
```
#Кортеж в строку
```

```
game_name = ('Breath', ' ', 'of', ' ', 'the',  
' ', 'Wild')
```

```
game_name = ' '.join(game_name)
```

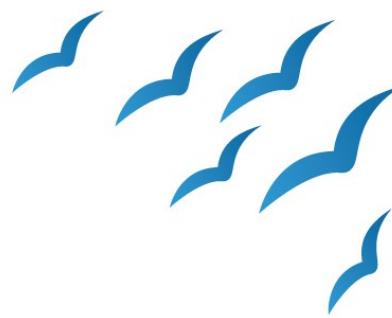
```
print(game_name)
```

```
Breath of the Wild
```



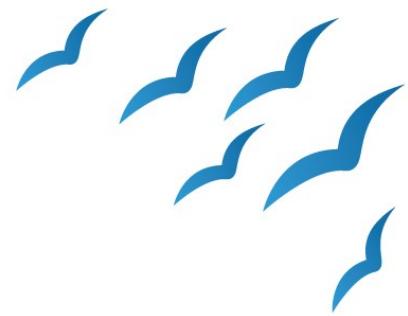
Преобразование tuple → list

```
dig_tuple = (1111, 2222, 3333)
print(dig_tuple)
> (1111, 2222, 3333)
dig_list = list(dig_tuple)
print(dig_list)
[1111, 2222, 3333]
```

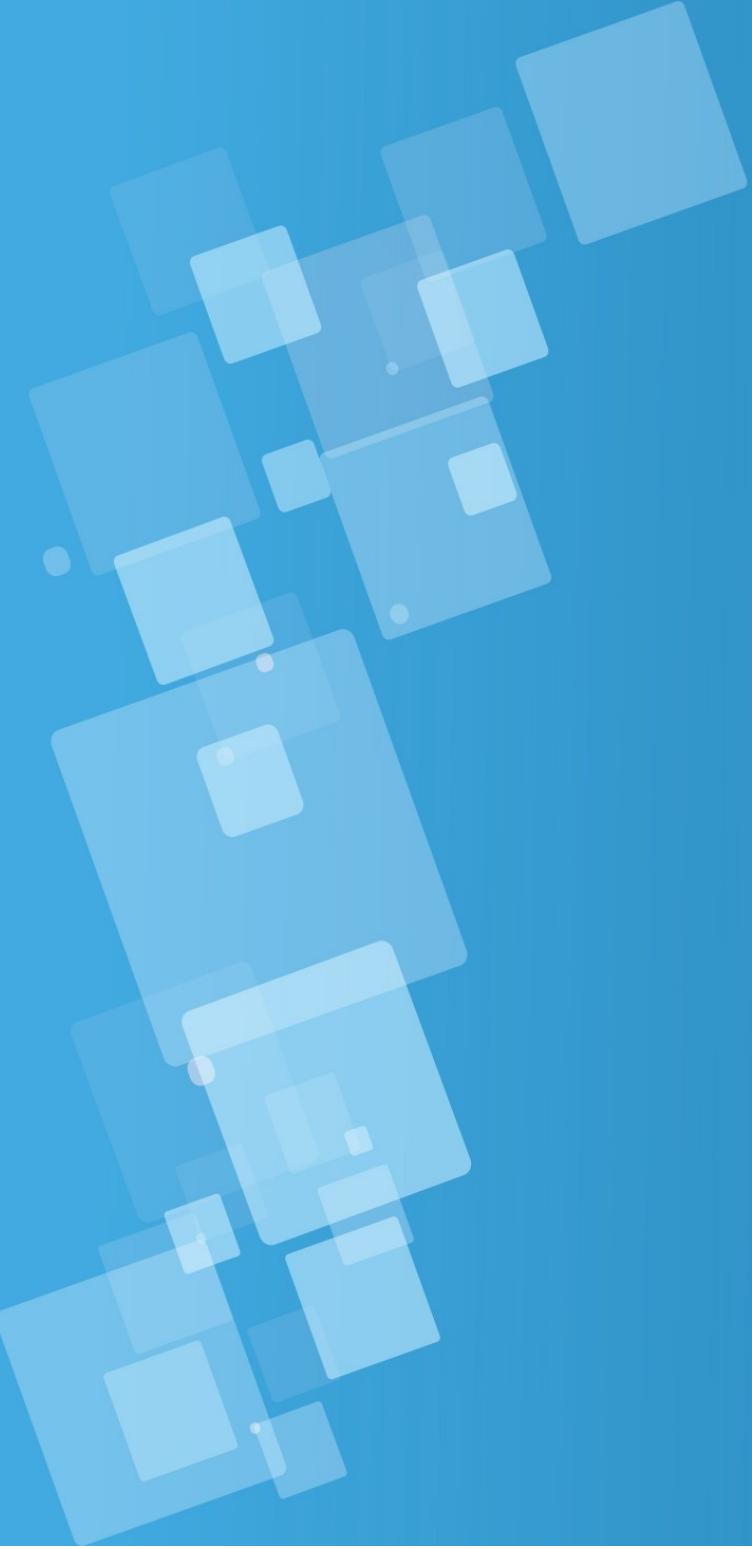


Преобразование tuple → dict

```
# Преобразование через генератор  
словарей  
  
person = (('name', 'Piter'), ('age', 100))  
p_dict = dict((x, y) for x, y in person)  
print(p_dict)  
{'name': 'Piter', 'age': 100}
```



Спасибо за внимание.



Функции

часть I

Понятие функции

Функция в Python - объект, принимающий аргументы и возвращающий значение.

Вход в функцию - это передача ей аргументов - данных, полученных во внешней части программы. Получив данные, функция должна их как-то обработать: выполнить некоторые действия, вычислить какое-то значение. Выход из функции - значение, вычисленное блоком кода данной функции и передаваемое во внешнюю часть программы. Входные данные называют параметрами, а выходные - возвращаемым значением. Впрочем, функция может и не принимать никаких параметров. Что принимает в качестве параметров и что возвращает функция в результате своей работы, определяет программист.

Роль функции в программировании

1. Сокращение кода

Код, который повторяется можно перенести в функцию и использовать её тогда, когда нужно выполнить код, который находится внутри этой функции.

2. Логическое разделение программы

Мы можем выделить определённое сложное действие (например перемножение матриц) в отдельную функцию, чтобы оно не мешалось в коде, даже если используем её один раз за всё время выполнения программы.

Определение функции

```
# объявление функции my_function()

def my_function([параметр1, параметр2, ...]):  
    # тело функции

    # возвращаемое значение
    return result # необязательно

# вызов функции
my_function([аргумент1, аргумент2, ...])

type(my_function)
<class 'function'> - еще один тип в Python
```

Пример:

```
#Определение функции:
```

```
def summ(x, y):  
    result = x + y  
return result
```

```
#вызов функции
```

```
a = 100
```

```
b = -50
```

```
answer = summ(a, b)
```

```
print(answer)
```

Вызов функции

Можно ли так вызвать функцию ?

```
summ(10)
```

```
def summ(x):  
    print(x)
```

NameError: name 'summ' is not defined

Что вернет функция без return:

#Определение функции:

```
def summ(x, y):  
    result = x + y
```

#вызов функции

```
answer = summ(100, -50)
```

```
print(answer) - ?
```

answer is None

True

Что вернет функция в отсутствии аргументов ?

#Определение функции:

```
def summ(x, y):  
    result = x + y  
    return result;
```

#вызов функции

```
answer = summ(100)
```

TypeError: summ() missing 1 required positional argument: 'y'

Параметры по умолчанию

Для некоторых параметров в функции можно указать значение по умолчанию, таким образом если для этого параметра не будет передано значение при вызове функции, то ему будет присвоено значение по умолчанию.

```
def premium(salary, percent=10):  
    p = salary * percent / 100  
    return p
```

```
result = premium(60000)  
print(result)
```

6000.0

```
print(premium(60000, 20))  
12000.0
```

Пустая функция

```
def empty(var1, var2):  
    pass
```

```
result = empty(8, 10)  
print(result)
```

None

Именованные параметры

Иногда происходит такая ситуация, что функция требует большое количество аргументов. Пример:

```
def my_func(arg1, arg2, arg3, arg4, arg5, arg6):  
    pass # оператор, если кода нет  
  
result = my_func(1, 2, 3, 5, 6)
```

В такой ситуации код не всегда удобно читать и тяжело понять, какие переменные к каким параметрам относятся.

```
result = my_func(arg2=2, arg1=1, arg4=4, arg5=5,  
arg3=3, arg6=6)
```

ФУНКЦИЯ С НЕОГРАНИЧЕННЫМ КОЛИЧЕСТВОМ ПОЗИЦИОННЫХ АРГУМЕНТОВ

***args** – произвольное число позиционных аргументов

```
def manyargs(var1, *args):  
    print(type(args))  
    print(args)
```

```
result = manyargs(4, 9, 1, 3, 3, 1)
```

```
<class 'tuple'>  
(9, 1, 3, 3, 1)
```

ФУНКЦИИ С НЕОГРАНИЧЕННЫМ КОЛИЧЕСТВОМ ИМЕНОВАННЫХ АРГУМЕНТОВ

****kwargs** – произвольное число именованных аргументов.

```
def manykwargs(**kwargs):
    print(type(kwargs))
    print(kwargs)

manykwargs(name='Piter', age=20)

<class 'dict'>
{'name': 'Piter', 'age': 20}
```

Можно ли мне по другому назвать параметр ****kwargs** ?

Можно, только вас никто не поймет !

Все вместе.

***args** – произвольное число позиционных аргументов

****kwargs** – произвольное число именованных аргументов.

```
def many_all(var1, *args, **kwargs):  
    print(var1)  
    print(args)  
    print(kwargs)
```

```
many_all(10, 34, 77, name='Piter', age=20)
```

```
10
```

```
(34, 77)
```

```
{'name': 'Piter', 'age': 20}
```

Можно ли мне по другому назвать параметр ****kwargs** ?

Можно, только вас никто не поймет !

Scope (область видимости)

Область видимости указывает интерпретатору, когда наименование (или переменная) видима. Другими словами, область видимости определяет, когда и где вы можете использовать свои переменные, функции и т.д. Если вы попытаетесь использовать что-либо, что не является в вашей области видимости, вы получите ошибку `NameError`. Python содержит три разных типа области видимости:

- * Локальная область видимости
- * Глобальная область видимости
- * Нелокальная область видимости (была добавлена в Python 3)

Локальная область видимости

Локальная область видимости (local) – это блок кода или тело любой функции Python или лямбда-выражения. Эта область Python содержит имена, которые вы определяете внутри функции.

```
x = 100
```

```
def doubling(y):
```

```
    z = y*y
```

```
doubling(x)
```

```
print(z)
```

```
NameError: name 'z' is not defined
```

Глобальная область видимости

Глобальная область видимости (global). В Python есть ключевое слово `global`, которое позволяет изменять изнутри функции значение глобальной переменной. Оно записывается перед именем переменной, которая дальше внутри функции будет считаться глобальной.

```
x = 100  
  
def doubling(y):  
    global x  
    x = y*y  
  
doubling(x)  
  
print(x)  
10000
```

Нелокальная область видимости

В Python 3 было добавлено новое ключевое слово под названием **nonlocal**. С его помощью мы можем добавлять переопределение области во внутреннюю область.

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

Если вы попробуете запустить этот код, вы получите ошибку `UnboundLocalError`, так как переменная `num` ссылается прежде, чем она будет назначена в самой внутренней функции.

Нелокальная область видимости

Добавим nonlocal в наш код:

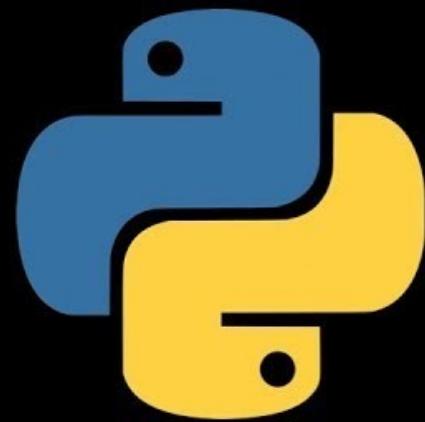
```
def counter():
    num = 0
    def incrementer():

        nonlocal num
        num += 1
        return num
    return incrementer
```

```
inc = counter()
```

```
inc()
```

1



PYTHON PROGRAMMING



Стандартные функции Input-Output

Файлы

Файлы - это просто массив байт на жёстком диске. В файлах мы можем хранить абсолютно любую информацию: текст, аудио, видео, изображения, исполняемый код и т.д. Логически принято делить файлы на:

1. Текстовые
2. Бинарные (двоичные)

Типы текстовых файлов

- .txt
- .CSV
- .xml
- .json
- .yaml

TXT

TXT — это формат файлов, который содержит текст, упорядоченный по строкам. Текстовые файлы отличаются от двоичных файлов, содержащих данные, не предназначенные для интерпретирования в качестве текста (закодированный звук или изображение).

Файл настроек Raspberry Pi config.txt

```
# Set stdv mode to PAL (as used in Europe)
sdtv_mode=2
# Force the monitor to HDMI mode so that sound will be sent
over HDMI cable
hdmi_drive=2
# Set monitor mode to DMT
hdmi_group=2
# Set monitor resolution to 1024x768 XGA 60Hz
(HDMI_DMT_XGA_60)
hdmi_mode=16
# Make display smaller to stop text spilling off the screen
overscan_left=20
overscan_right=12
overscan_top=10
overscan_bottom=10
```

CSV

CSV (comma-separated value) - это формат представления табличных данных (например, это могут быть данные из таблицы или данные из БД).

В этом формате каждая строка файла - это строка таблицы. Несмотря на название формата, разделителем может быть не только запятая.

И хотя у форматов с другим разделителем может быть и собственное название, например, TSV (tab separated values), тем не менее, под форматом CSV понимают, как правило, любые разделители.

Пример файла cvs

1.04.2022, 1 апреля, 13.00, "Открытие выставки"

1.04.2022, 1 апреля, 18.00, "Показ музыкальной комедии"

2.04.2022, 2 апреля, 16.00, "Концертная программа «Мелодия любви»"

5.04.2022, 5 апреля, 18.00, "Показ боевика"

.xml

XML — текстовый формат, предназначенный для хранения структурированных данных . XML разрабатывался как язык с простым формальным синтаксисом, удобный для создания и обработки документов как программами, так и человеком, с акцентом на использование в Интернете. Язык называется расширяемым, поскольку он не фиксирует разметку, используемую в документах: разработчик волен создать разметку в соответствии с потребностями к конкретной области, будучи ограниченным лишь синтаксическими правилами языка.

Файл workspace.xml в папки .idea

```
▼<component name="TaskManager">
  ▼<task active="true" id="Default" summary="Default task">
    <changelist id="809dbb00-5478-42ba-ae09-75e83407f4b8" name="Ch
    <created>1649147023499</created>
    <option name="number" value="Default"/>
    <option name="presentableId" value="Default"/>
    <updated>1649147023499</updated>
  </task>
  ▼<task id="LOCAL-00001" summary="Добавилена диаграмма Mermaid">
    <created>1649153680257</created>
    <option name="number" value="00001"/>
    <option name="presentableId" value="LOCAL-00001"/>
    <option name="project" value="LOCAL"/>
    <updated>1649153680257</updated>
  </task>
  ▼<task id="LOCAL-00002" summary="Добавилена диаграмма Mermaid">
    <created>1649153785124</created>
    <option name="number" value="00002"/>
    <option name="presentableId" value="LOCAL-00002"/>
    <option name="project" value="LOCAL"/>
    <updated>1649153785124</updated>
  </task>
  ▼<task id="LOCAL-00003" summary="Hotfix">
    <created>1649154138050</created>
    <option name="number" value="00003"/>
    <option name="presentableId" value="LOCAL-00003"/>
    <option name="project" value="LOCAL"/>
    <updated>1649154138050</updated>
  </task>
  ▼<task id="LOCAL-00004" summary="Hotfix">
    <created>1649154793842</created>
    <option name="number" value="00004"/>
```

.json

JSON (JavaScript Object Notation) - простой формат обмена данными, удобный для чтения и написания как человеком, так и компьютером. Он основан на подмножестве языка программирования JavaScript, определенного в стандарте ECMA-262 3rd Edition - December 1999. ... Эти свойства делают JSON идеальным языком обмена данными. JSON основан на двух структурах данных: Коллекция пар ключ/значение.

Файл package.json для npm

```
{  
  "name": "cdpo",  
  "version": "1.0.0",  
  "private": true,  
  "scripts": {  
    "dev": "nuxt",  
    "build": "nuxt build",  
    "start": "nuxt start",  
    "generate": "nuxt generate"  
  },  
  "dependencies": {  
    "amqplib": "^0.8.0",  
    "core-js": "^3.19.3",  
    "nuxt": "^2.15.8",  
    "vue": "^2.6.14",  
    "vue-server-renderer": "^2.6.14",  
    "vue-template-compiler": "^2.6.14",  
    "vuetify": "^2.6.1",  
    "webpack": "^4.46.0"  
  },  
  "devDependencies": {  
    "@nuxtjs/moment": "^1.6.1",  
    "@nuxtjs/vuetify": "^1.12.3"  
  }  
}
```

.yaml

YAML — это формат файла, обычно используемый для сериализации данных. Существует множество проектов, использующих файлы YAML для настройки, таких как Docker-compose, pre-commit, TravisCI, AWS Cloudformation, ESLint, Kubernetes, Ansible и многие другие.

Конфигурационный файл swagger

```
1. definitions:  
2.   CatalogItem:  
3.     type: object  
4.     properties:  
5.       id:  
6.         type: integer  
7.         example: 38  
8.       title:  
9.         type: string  
10.        example: T-shirt  
11.       required:  
12.         - id  
13.         - title
```

Открытие файла

Работать с файлами можно тремя способами:

1. Читать из файла
2. Записывать в файл
3. Дозаписывать в файл

Открытие файла

Прежде, чем работать с файлом, его надо открыть. Для этой задачи есть встроенная функция open:

```
f = open("text.txt", encoding="utf-8")
```

Результатом работы функция open возвращает специальный объект, который позволяет работать с файлом (файловый дискриптор)

Можно ли указать "utf-8" без **encoding=** ?

Синтаксис функции open()

```
fp = open(file, mode='r', buffering=-1, encoding=None,  
         errors=None, newline=None, closefd=True, opener=None)
```

Параметры:

file – абсолютное или относительное значение пути к файлу или файловый дескриптор открываемого файла.

mode – необязательно, строка, которая указывает режим, в котором открывается файл. По умолчанию 'r'.

buffering – необязательно, целое число, используемое для установки политики буферизации.

encoding – необязательно, кодировка, используемая для декодирования или кодирования файла.

errors – необязательно, строка, которая указывает, как должны обрабатываться ошибки кодирования и декодирования. Не используется в бинарном режиме

newline – необязательно, режим перевода строк. Варианты: None, '\n', '\r' и '\r\n'. Следует использовать только для текстовых файлов.

closefd – необязательно, bool, флаг закрытия файлового дескриптора.

opener – необязательно, пользовательский объект, возвращающий открытый дескриптор файла.

У функции **open()** много параметров, нам пока важны 3 аргумента: первый, это имя файла. Путь к файлу может быть относительным или абсолютным.

Второй аргумент - это режим, *mode*, в котором мы будем открывать файл. Режим обычно состоит из двух букв, первой является тип файла - текстовый или бинарный, в котором мы хотим открыть файл, а второй указывает, что именно мы хотим сделать с файлом.

Третий аргумент - кодировка файла

Первая буква режима:

"*b*" - открытие в двоичном режиме.

"*t*" - открытие в текстовом режиме (является значением по умолчанию).

Второй буква режима:

"*r*" - открытие на чтение (является значением по умолчанию).

"*w*" - открытие на запись, содержимое файла удаляется, если файла не существует, создается новый.

"*x*" - эксклюзивное создание(открытие на запись), бросается исключение `FileExistsError`, если файл уже существует.

"*a*" - открытие на дозапись, информация добавляется в конец файла.

"*+*" - открытие на чтение и запись

Примеры

```
# Режим "w" открывает файл только для записи.  
Перезаписывает файл, если файл существует. Если файл  
не существует, создает новый файл для записи.
```

```
f = open("text.txt", mode="w"  encoding="utf-8")
```

```
# Открывает файл в бинарном режиме для записи и  
чтения. Перезаписывает существующий файл, если файл  
существует. Если файл не существует, создается новый  
файл для чтения и записи.
```

```
f = open("music.mp3", mode="wb+")
```

По всем режимам см. [документацию open\(\)](#)

Закрыть файл

После того как вы сделали всю необходимую работу с файлом - его следует закрыть.

```
f = open("text.txt", encoding="utf-8")  
# какие-то действия  
f.close()
```

Кто ограничивает максимальное количество открытых файловых дискрипторов ?

`ulimit -Hn`

Чтение файла

Теперь мы хотим прочитать из него информацию. Для этого есть несколько способов, но большого интереса заслуживают лишь два из них. Первый - метод **read**, читающий весь файл целиком, если был вызван без аргументов, и n символов, если был вызван с аргументом (целым числом n).

```
f = open("text.txt", "rt")  
  
print(f.read(5))  
print(f.read(5))  
print(f.read(4))  
print(f.read())  
  
f.close()
```

Чтение файла

Теперь мы хотим прочитать из него информацию. Для этого есть несколько способов, но большого интереса заслуживают лишь два из них. Первый - метод **read**, читающий весь файл целиком, если был вызван без аргументов, и n символов, если был вызван с аргументом (целым числом n).

```
f = open("text.txt", "rt")  
  
print(f.read(5))  
print(f.read(5))  
print(f.read(4))  
print(f.read())  
  
f.close()
```

Функция `readlines()`

Файлы можно читать не только целиком или посимвольно, но и построчно. Для этого у объекта файла есть метод `readlines` который возвращает список из строк файла.

```
f = open("text.txt", "rt")
print(f.readlines())
f.close()
```

Обратите внимание, что каждая строка в списке имеет в конце символ `\\n`.

Функция `readline()`

Функция `'readlines'` загружает все строки целиком и хранит их в оперативной памяти, что может быть очень накладно, если файл занимает много места на жёстком диске. Можно читать файл построчно с помощью функции `'readline'`

```
f = open("text.txt", "rt")
print(f.readline())
print(f.readline())
f.close()
```

Также обратите внимание, что возвращённые строки имеют в конце символ `'\n'`.

Итерирование файла

Ещё один способ прочитать файл построчно – использовать файл как итератор. Такой вариант считается самым оптимизированным

```
f = open("text.txt")  
for line in f:  
    print(line)  
  
f.close()
```

Запись

(*) rec

Теперь рассмотрим запись в файл. Для того чтобы можно было записывать информацию в файл, нужно открыть файл в режиме записи. Для записи в файл используется функция `write`. При открытии файла на запись из него полностью удаляется предыдущая информация.

```
f = open("text.txt", "wt")  
f.write("New string")  
f.write("Another string")  
f.close()
```

Если вы откроете файл в текстовом редакторе, то увидите, что строки "New string" и "Another string" склеились. Так произошло, потому что между ними нет символа перевода строки.

Также в файлах, открытых на запись, есть метод `writelines`, который позволяет записать несколько строк в файл

```
f = open("text.txt", "wt")
lines = [
    "New string\n",
    "Another string\n",
]
f.writelines(lines)
f.close()
```

Дозапись

Если нужно записать в конец файла какую-то информацию, то можно сделать это открыв файл в режиме дозаписи. Все методы, доступные в режимах записи также доступны в режиме дозаписи.

```
f = open("text.txt", "at")  
  
f.write("First string\n")  
lines = [  
    "Second string\n",  
    "Third string\n",  
]  
f.writelines(lines)  
  
f.close()
```

Запись с возможностью чтения

Иногда нужно открыть файл с возможностью и записи, и чтения.
В Python есть два режима:

- * Запись с возможностью чтения ("w+")
- * Чтение с возможностью записи ("r+")

На первый взгляд кажется, что они ничем не отличаются, но это не так.

При открытии файла на запись с возможностью чтения из файла полностью удаляется вся информация. Вы можете записывать и читать из файла одновременно.

Пример.

(*) rec

```
f = open("text.txt", "w+t")
```

```
print(f.read())
f.write("Hello\n")
print(f.read())
```

```
f.close()
```

Чтение с возможностью записи

При открытии файла на чтение с возможностью записи файл не перезаписывается.

```
f = open("text.txt", "r+t")
```

```
print(f.read(1))
f.write("A")
f.read()
```

```
f.close()
```

При записи символы в файле затирают символы, идущие следом, как если бы вы в текстовом редакторе перевели указатель в середину текста, нажали *insert* и начали бы печатать.

PEP8

В PEP8 описано, что в конце файла с кодом всегда нужно оставлять пустую строку. Это правило кажется надуманным, но сейчас мы знаем, что в любой файл, в котором последним символом стоит перевод строки можно программно дозаписать любую строку и она не склеится с последней строкой в файле.

Управляющие символы

\n	(newline) перевод каретки на следующую строку
\r	(return) перевод каретки на в начало текущей строки
\t	(tab) табуляция (отступ, красная строка)
\b	(backspace) перевод каретки на один символ назад

Указатель позиции

При чтении файла функция `read` читает символы друг за другом, а при записи в файл все строки (строки байт) записываются последовательно друг за другом. Это поведение объясняется тем, что python хранит специальный указатель, позиция этого указателя говорит, с какого места читать из файла или писать в файл.

Независимо от того в каком режиме открыт файл у каждого объекта файла есть методы `tell` и `seek`. Метод `tell` возвращает целое число – позицию, где сейчас находится указатель. Метод `seek` принимает целое число и переносит указатель в указанную позицию. Например, передвинуть указатель на две позиции вперёд можно следующим образом

```
position = f.tell()  
f.seek(position + 2)
```

file.seek(offset[, whence])

Параметры:

file - объект файла

offset - int байтов, смещение указателя чтения/записи файла.

whence - int, абсолютное позиционирование указателя.

Возвращаемое значение:

целое число int, новая позиция указателя.

Описание:

Метод файла file.seek() устанавливает текущую позицию в байтах offset для указателя чтения/записи в файле file.

Аргумент whence является необязательным и по умолчанию равен 0.

Может принимать другие значения:

0 - означает, что нужно сместить указатель на offset относительно начала файла.

1 - означает, что нужно сместить указатель на offset относительно относительно текущей позиции.

2 - означает, что нужно сместить указатель на offset относительно конца файла.

Пример 1

```
# Начать чтение с 3 символа строки  
f = open('testFile.txt', 'r')  
f.seek(3)  
print(f.read())
```

Пример 2

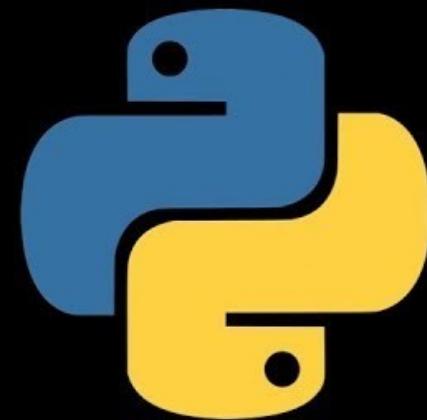
```
>>> text = b'This is 1st line\nThis is 2nd line\nThis is 3rd line\n'
>>> fp = open('foo.txt', 'bw+')
>>> fp.write(text)
# 51

>>> fp.seek(20, 0)
# 20
>>> fp.read(10)
# b's is 2nd l'

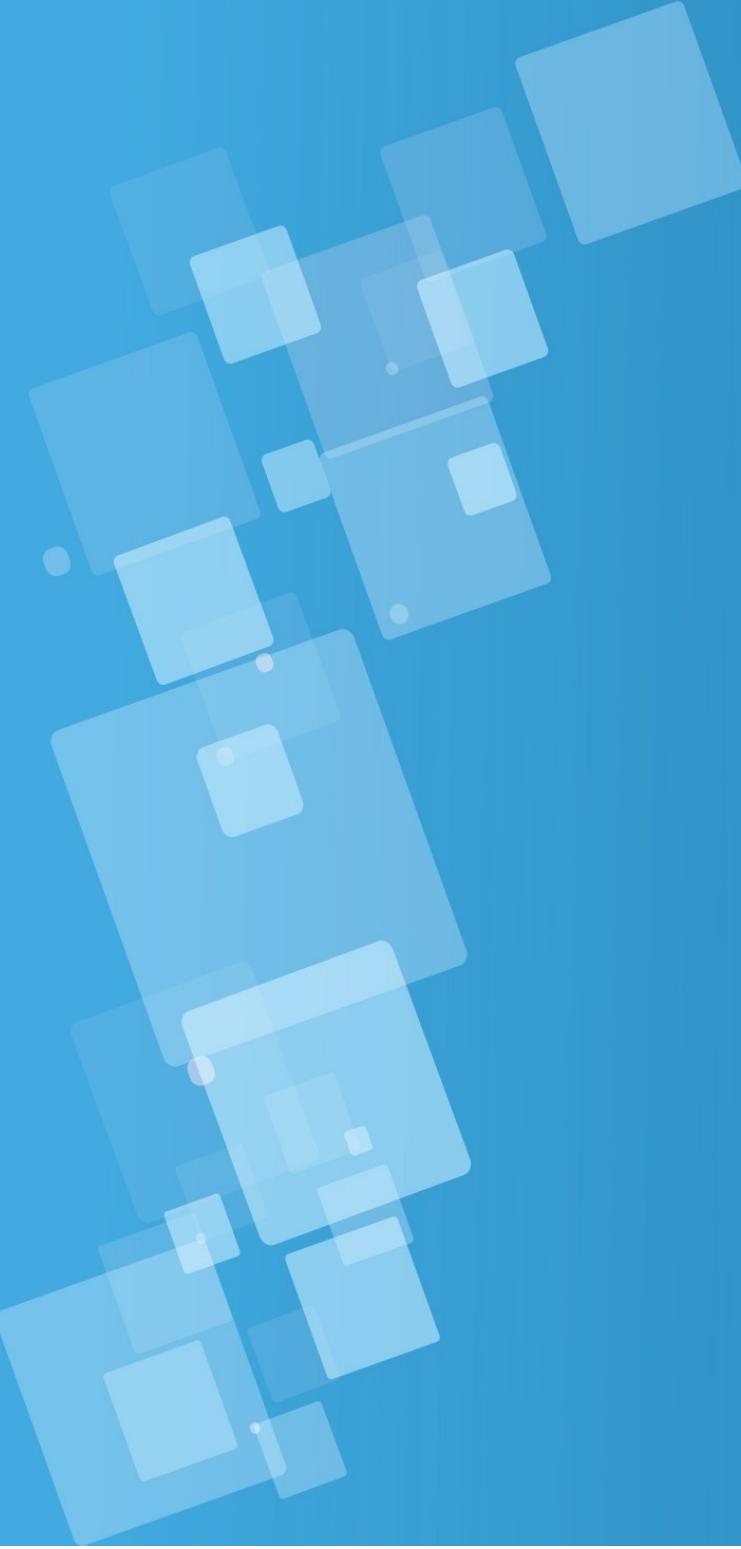
>>> fp.seek(10, 1)
# 40
>>> fp.read(10)
# b's 3rd line'

>>> fp.seek(-11, 2)
# 40
>>> fp.read(10)
# b's 3rd line'

>>> fp.close()
```



PYTHON PROGRAMMING



Стандартные функции

методы строк
дата и время

Работа с текстом

954



пл. 11, кв. 1656

0

5 см



Работа со строками – неотъемлемая часть создания практически любого приложения, где используется текст, и язык программирования Python предоставляет широкие возможности работы с такими данными.

Форматирование строковых значений

I Способ основан на модели printf языка С

```
>>> name = "Alex"  
>>> 'Hello, %s' % name  
'Привет, Alex'
```

1. '%d', '%i', '%u — десятичное число;
2. '%c' — символ, точнее строка из одного символа или число — код символа;
3. '%r' — строка (литерал Python);
4. '%s' — строка.

В подстановки используется несколько аргументов, в правой части будет кортеж со строками:

```
>>> '%d %s, %d %s' % (6, 'bananas', 10, 'lemons')
'6 bananas, 10 lemons'
```

Префикс `r` обозначает неформатируемые (или необрабатываемые) строки, в которых подавляется действие символов экранирования. Такие строки очень удобны, для хранения путей к файлам в Windows, например:

```
str = r'D:\\мои документы\\книги\\Лутц.pdf'
>>> 'Путь к файлу: %r' % str
```

Позиционирование аргументов по имени. В правой части будет словарь с ключами:

```
>>> '%(count_)d %(fruit)s' % {'fruit': 'bananas',  
'count_': 100}
```

II Способ совпадает с выражениями форматирования строк
`str.format()`

```
>>> name = "Alex"  
>>> 'Hello, {}' .format(name)  
'Привет, Alex'
```

Позиция подстановки может быть изменена

```
>>> name = "Alex"  
>>> 'Hello, {0} {1} {2}' .format(name, "Bob", "Lulu")  
'Привет, Alex'
```

III способ форматирования строк появился в Python 3.6
f-строки

```
>>> name = "Alex"  
>>> f'Hello, {name}'  
'Привет, Alex'
```

Также можно указать тип при подстановке. Литералы букв аналогичны способу 1.

```
>>> name = "Alex"  
>>> age = 16  
>>> f'Hello, {name:#s  age:16 }'
```

Способ допускает возможность встраивать выражения
x = y = 5

F'Сумма чисел x и y:, {x + y:#d}'

x = y = 5.4

F'Сумма чисел x и y:, {x + y:#f}'

Функции для работы со строками

`str(n)` – преобразование числового или другого типа к строке;

`len(s)` – длина строки;

`chr(s)` – получение символа по его коду ASCII;

`ord(s)` – получение кода ASCII по символу;

Методы для работы со строками

find(s, start, end) – возвращает индекс первого вхождения подстроки в s или -1 при отсутствии. Поиск идет в границах от start до end;

rfind(s, start, end) – аналогично, но возвращает индекс последнего вхождения;

replace(s, new) – меняет последовательность символов s на новую подстроку new;

split(x) – разбивает строку на подстроки при помощи выбранного разделителя x;

join(x) – соединяет строки в одну при помощи выбранного разделителя x;

strip(s) – убирает пробелы с обеих сторон;

lstrip(s), **rstrip**(s) – убирает пробелы только слева или справа;

lower() – перевод всех символов в нижний регистр;

upper() – перевод всех символов в верхний регистр;

capitalize() – перевод первой буквы в верхний регистр, остальных – в нижний.

isdigit() – состоит ли строка из цифр

isalpha() – состоит ли строка из букв

isalnum() – состоит ли строка из цифр или букв

find(s, start, end)

Метод str.find() возвращает индекс первого совпадения подстроки sub в строке str, где подстрока или символ sub находится в пределах среза str[start:end].

```
>>> x = 'раз два три раз два три раз'
```

```
>>> x.find('раз')
```

```
# 0
```

```
>>> x.find('раз', 10, 23)
```

```
# 12
```

```
>>> x.find('раз', -12)
```

```
# 24
```

```
>>> x = 'раз два три раз два три раз'
```

```
>>> x.find('четыре')
```

```
# -1
```

rfind(s, start, end)

```
>>> txt = "Mi casa, su casa."  
>>> x = txt.rfind("casa")  
>>> print(x)  
12
```

```
txt = "Hello, welcome to my world."  
x = txt.rfind("e", 5, 10)  
print(x)  
8
```

index(s, start, end)

Метод выдает индекс первого вхождения.

```
txt = "Hello, welcome to my world."  
x = txt.index("welcome")  
print(x)
```

В отличии от find выдаст ошибку

```
txt = "Hello, welcome to my world."  
x = txt.index("goodbay")  
print(x)
```

ValueError: substring not found

replace(oldvalue, newvalue, count)

Параметры:

oldvalue – строка для поиска

newvalue – строка замены

count – сколько вхождений заменить, по умолчанию all()

```
txt = "I like bananas"
```

```
x = txt.replace("bananas", "apples")
```

```
print(x)
```

Могу ли я сделать так ?

```
txt[0] = "Y"
```

replace(oldvalue, newvalue, count)

Что произойдет ?

```
txt = "I like bananas"  
x = txt.replace("anas", "apples")  
print(x)
```

'I like banapples'

split(separator, maxsplit)

Параметры:

separator – разделитель используемый для разбивки
maxsplit – определяет кол-во операций разделений.
По умолчанию все вхождения.

```
txt = "hello, my name is Peter, I am 26 years old"  
x = txt.split(", ")  
print(x)  
['hello', 'my name is Peter', 'I am 26 years old']
```

split(separator, maxsplit)

```
txt = "apple#banana#cherry#orange"
```

```
x = txt.split("#", 1)
```

```
print(x)
```

```
['apple', 'banana#cherry#orange']
```

join(iterable)

```
myTuple = ("John", "Peter", "Vicky")
x = "#".join(myTuple)
print(x)
John#Peter#Vicky
```

```
myDict = {"name": "John", "country": "Norway"}
mySeparator = "_"
x = mySeparator.join(myDict)
print(x)
'name_country'
```

strip(characters)

Параметры:

Characters – опциональный, устанавливает символы для удаления из текста

```
# удаление пробелов
```

```
>>> text = " test "
```

```
>>> text.strip()
```

```
'test'
```

```
txt = " , , , , rrttgg.....banana....rrr"
```

```
x = txt.strip(", .grt")
```

```
print(x)
```

```
banana
```

lstrip(characters) rstrip(characters)

Параметры:

characters – опциональный, устанавливает символы для удаления из текста

удаление пробелов слева

```
>>> text = "...test..."
```

```
>>> text.lstrip(".")
```

```
'test...'
```

```
>>> text = "...test..."
```

```
>>> text.rstrip(".")
```

```
'...test'
```

lower() , upper()

Параметры:

```
txt = "Hello my FRIENDS"
```

```
x = txt.lower()
```

```
print(x)
```

```
Hello my friends
```

```
txt = "Hello my friends"
```

```
x = txt.upper()
```

```
print(x)
```

```
HELLO MY FRIENDS
```

capitalize()

Параметры:

```
txt = "python is FUN!"  
x = txt.capitalize()  
print (x)  
Python is fun!
```

isdigit() , isalpha() , isalnum()

Параметры:

```
block = "1024"
```

```
block.isdigit()
```

True

```
era = "XXI Centure"
```

```
era.isalpha()
```

False

```
era = "2022Centure"
```

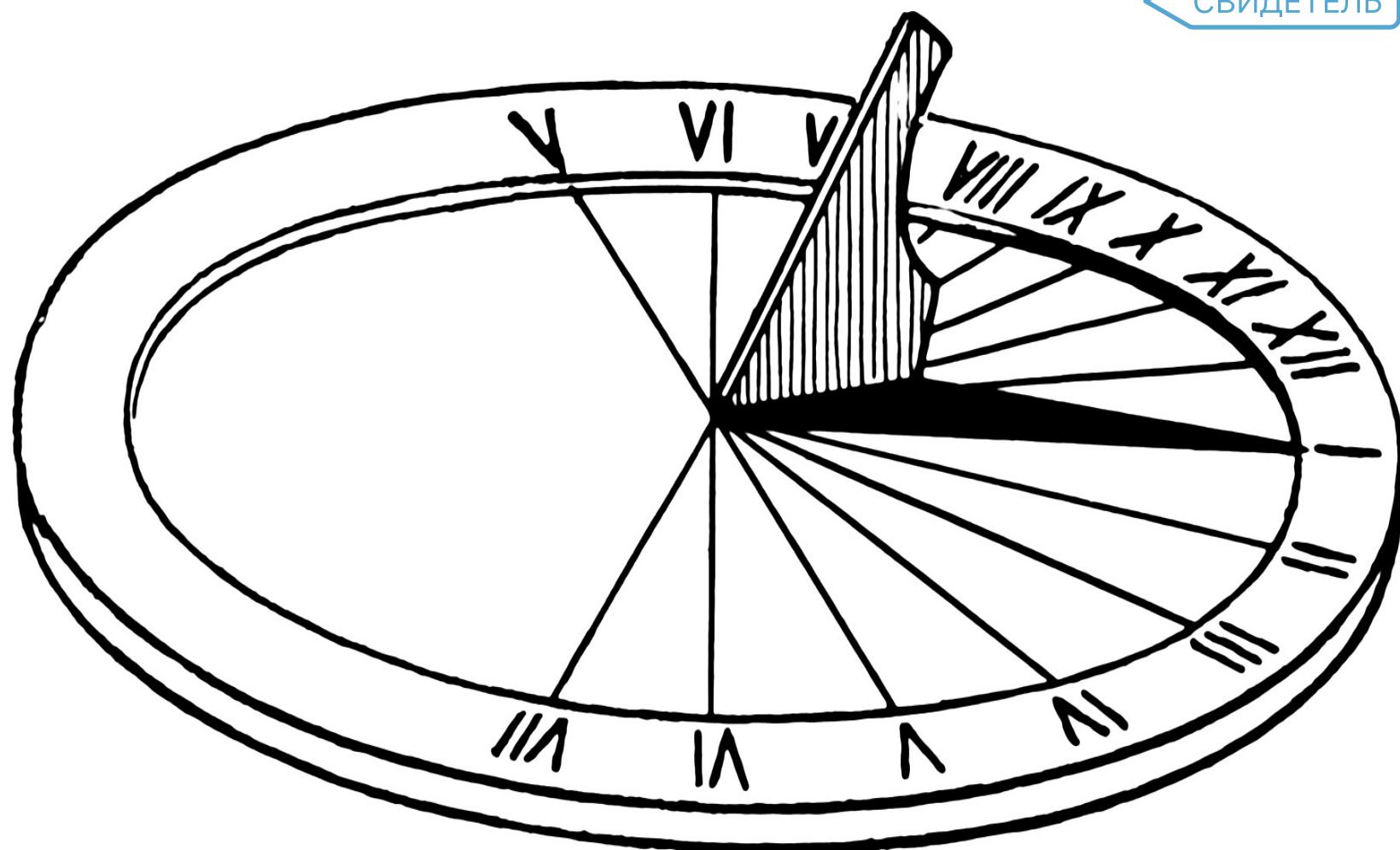
```
era.isalnum()
```

True

Преобразование строки в дату.

```
# Воспользуемся библиотекой datetime методом strftime  
  
>>> from datetime import datetime  
  
>>> print(datetime.strptime('22 04 2020 19:33', '%d  
%m %Y %H:%M'))  
2020-04-22 19:33:00
```

Работа с датой и временем



СВИДЕТЕЛЬ

Получение текущей даты и времени.

Одним из классов, определенных в модуле `datetime`, является класс `datetime`. После импортирования класса мы использовали метод `now()` для создания объекта `datetime`, содержащего текущие локальные дату и время.

```
from datetime import datetime  
datetime_object = datetime.now()  
print(datetime_object)
```

2021-03-22 12:04:13.536031

Создание даты и времени

```
from datetime import datetime
a = datetime(2017, 11, 28, 23, 55, 59, 342380)
print("year =", a.year)
print("month =", a.month)
print("hour =", a.hour)
print("minute =", a.minute)
print("timestamp =", a.timestamp())
```

Получение текущей даты.

В этой программе мы использовали метод `today()`, определенный в классе `date`, чтобы получить объект `date`, содержащий текущую локальную дату.

```
from datetime import date  
today = date.today()  
print("Current date =", today)
```

2022-04-22

Конструирование даты

```
import datetime
dt = datetime.date(2020, 6, 29)
print(dt)
2020-06-29
# получение значений
print("Current year:", dt.year)
print("Current month:", dt.month)
print("Current day:", dt.day)
```

Получение даты из метки времени (timestamp).

Термин timestamp употребляется для обозначения POSIX-времени – количества секунд, прошедшего с 00:00:00 UTC 1 января, 1970 года. Вы можете преобразовать метку времени в дату при помощи метода fromtimestamp()

Создание метки:

```
import time  
  
from datetime import date  
  
now = time.time()
```

Конструирование объекта времени date

```
timestamp = date.fromtimestamp(now)  
  
print("Date =", timestamp)
```

2022, 4, 22

Форматирование даты

```
from datetime import datetime
now = datetime.now()
t = now.strftime("%H:%M:%S")
print("time:", t)
time: 15:00:24
```

```
s1 = now.strftime("%m/%d/%Y, %H:%M:%S")
print("s1:", s1)
s1: 04/22/2022, 15:00:24
```

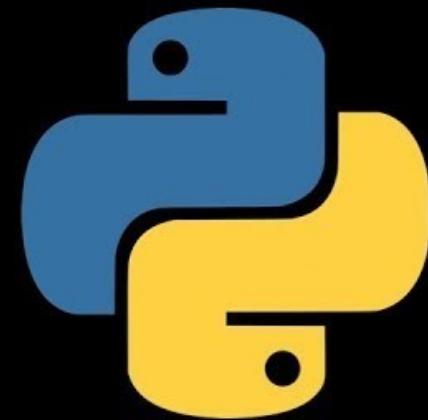
```
s2 = now.strftime("%d/%m/%Y, %H:%M:%S")
# dd/mm/YY H:M:S format
print("s2:", s2)
```

Основные коды для определения формата:

- %Y – год [0001, ..., 2018, 2019, ..., 9999]
- %m – месяц [01, 02, ..., 11, 12]
- %d – день [01, 02, ..., 30, 31]
- %H – час [00, 01, ..., 22, 23]
- %M – минута [00, 01, ..., 58, 59]
- %S – секунда [00, 01, ..., 58, 59]



Спасибо за внимание!



PYTHON PROGRAMMING



Изолированное
окружение.
Пакеты, модули



Модули и пакеты

Система модулей позволяет вам логически организовать ваш код на Python. Группирование кода в модули значительно облегчает процесс написания и понимания программы. Говоря простым языком, модуль в Python это **просто файл**, содержащий код на Python. Каждый модуль в Python может содержать **переменные, объявления классов и функций**. Кроме того, в модуле может находиться **исполняемый код**.

Команда **import**



Вы можете использовать любой питоновский файл как модуль в другом файле, выполнив в нем команду **import**. Команда **import** в Python обладает следующим синтаксисом:

```
import module_1[, module_2[, ... module_N]]
```

Когда интерпретатор Python встречает команду **import**, он импортирует этот модуль, если он присутствует в пути поиска Python. Путь поиска Python это список директорий, в которых интерпретатор производит поиск перед попыткой загрузить модуль.



Например, чтобы использовать модуль math следует написать:

```
import math
```

```
# Используем функцию sqrt из модуля math
```

```
print(math.sqrt(9))
```

```
# Печатаем значение переменной pi, определенной в math
```

```
print(math.pi)
```

Важно знать! Модуль загружается лишь однажды, независимо от того, сколько раз он был импортирован. Это препятствует циклическому выполнению содержимого модуля.

from module import var

Выражение `from ... import ...` не импортирует весь модуль, а только предоставляет доступ к конкретным объектам, которые мы указали.

```
# Импортируем из модуля math функцию sqrt
from math import sqrt

# Выводим результат выполнения функции sqrt.
# Обратите внимание, что нам больше незачем указывать
имя модуля

print (sqrt(144))

# Но мы уже не можем получить из модуля то, что не
импортировали !!!

print (pi) # Выдаст ошибку
```



```
from module import var, func, class
```

Импортировать из модуля объекты можно через запятую.

```
from math import pi, sqrt  
print(sqrt(121))  
print(pi)  
print(e)
```

from *module* **import** *

В Python так же возможно импортировать всё (переменные, функции, классы) за раз из модуля, для этого используется конструкция **from ... import ***

```
from math import *
```

```
# Теперь у нас есть доступ ко всем функция и  
переменным, определенным в модуле math
```

```
print(sqrt(121))  
print(pi)  
print(e)
```

Не импортируются объекты с **var**

Повторяющиеся названия перезаписываются. Такое поведение нужно отслеживать при импорте нескольких модулей.



```
import module_1 [ ,module_2 ]
```

За один раз можно импортировать сразу несколько модулей, для этого их нужно перечислить через запятую после слова `import`

```
import math, os  
print(math.sqrt(121))  
print(os.env)
```

import module as my_alias

Если вы хотите задать псевдоним для модуля в вашей программе, можно воспользоваться вот таким синтаксисом

```
import math as matan  
print(matan.sqrt(121))
```

Местонахождение модулей в Python

Когда вы импортируете модуль, интерпретатор Python ищет этот модуль в следующих местах:

- Директория, в которой находится файл, в котором вызывается команда импорта
- Если модуль не найден, Python ищет в каждой директории, определенной в консольной переменной **PYTHONPATH**.
- Если и там модуль не найден, Python проверяет путь заданный по умолчанию

Путь поиска модулей сохранен в системном модуле `sys` в переменной `path`. Переменная `sys.path` содержит все три вышеописанных места поиска модулей.

Получение списка всех модулей Python установленных на компьютере

Для того, чтобы получить список всех модулей, установленных на вашем компьютере достаточно выполнить команду:

```
>>>help ("modules")
```

Через несколько секунд вы получите список всех доступных модулей.

Создание своего модуля в Python

Чтобы создать свой модуль в Python достаточно сохранить ваш скрипт с расширением .py. Теперь он доступен в любом другом файле. Например, создадим два файла: module_1.py и module_2.py и сохраним их в одной директории. В первом запишем:

```
# module_1.py
def hello():
    print("Hello from module_1")
```

А во втором вызовем эту функцию:

```
# module_2.py
from module_1 import hello
hello()
```

Пакеты модулей в Python

Отдельные файлы-модули с кодом на Python могут объединяться в пакеты модулей. Пакет это директория (папка), содержащая несколько отдельных файлов-скриптов.

Например, имеем следующую структуру:

my_file.py

my_package

__init__.py

inside_file.py

В файле `inside_file.py` определена некая функция `foo`. Тогда чтобы получить доступ к функции `foo`, в файле `my_file` следует выполнить следующий код:

```
from my_package.inside_file import foo
```

Функция `dir()`

Встроенная функция `dir()` возвращает отсортированный список строк, содержащих все имена, определенные в модуле.

```
# на данный момент нам доступны лишь встроенные функции
```

dir()

```
# импортируем модуль math
```

```
import math
```

```
# теперь модуль math в списке доступных имен
```

dir()

```
# получим имена, определенные в модуле math
```

```
dir(math)
```

Менеджер пакетов pip (Python Package Index)



Установка pip

Начиная с Python версии 3.4, pip поставляется вместе с интерпретатором Python. Метод универсален и подходит для любой операционной системы, если в ней уже установлена какая-либо версия Python

Открыть консоль (терминал)

Скачать файл get-pip.py:

```
wget https://bootstrap.pypa.io/get-pip.py
```

Установить pip:

```
python3 get-pip.py
```

Использование pip

Самый распространённый способ использования pip - это через консоль (терминал). Чтобы использовать pip, в консоли нужно вызвать команду pip для Python2 или pip3 для Python3. Для того, чтобы узнать какие команды есть в pip нужно вызвать pip3 -help:

Usage:

pip3 <command> [options]

Commands:

install	Install packages.
download	Download packages.
uninstall	Uninstall packages.
freeze	Output installed packages in requirements format.
list	List installed packages.
show	Show information about installed packages.
check	Verify installed packages have compatible dependencies.
config	Manage local and global configuration.
search	Search PyPI for packages.
wheel	Build wheels from your requirements.
hash	Compute hashes of package archives.
completion	A helper command used for command completion.
help	Show help for commands.

install

Команда `install` позволяет установить какой-либо пакет.

```
pip3 install Flask==2.1
```

После `Flask` мы также указали версию пакета, которую мы хотим установить. Это необязательно, если мы не укажем версию, то установится самая последняя версия пакета, которая присутствует в репозитории.

```
pip3 install Flask
```

Также можно указывать ограничения на версии, к примеру, что хотим установить `Django` не старше версии

```
pip3 install Flask > 2.1
```



pip install -r *reqfile.txt*

Установка пакетов перечисленных в файле

```
pip3 install -r requirements.txt
```

Файл requirements.txt

```
Flask==2.0.2
Flask-JWT-Extended==4.3.1
Flask-RESTful==0.3.9
Flask-SQLAlchemy==2.5.1
passlib==1.7.4
pymongo==4.0.1
Werkzeug==2.0.2
```

Установленные пакеты будут храниться в папке
/python3.X/site-packages

Импортирование в скрипте

После установки пакета его можно импортировать в скрипт.

```
from flask import Flask  
  
app = Flask(__name__)  
  
@app.route("/")  
  
def hello_world():  
  
    return "<p>Hello, World!</p>"
```

Понижение версии --force-reinstall

А если пакет уже установлен и вы хотите понизить его версию добавьте **--force-reinstall** вот так:

```
pip install 'stevedore>=1.3.0,<1.4.0' --force-reinstall
```

Проблемы **--no-cache-dir**

Иногда ранее установленная версия кэшируется.

При установки новой указанной версии пакета

```
pip install pillow==5.2.0
```

pip возвращает следующее:

```
Требование уже выполнено: pillow==5.2.0 in  
/home/ubuntu/anaconda3/lib/python3.6/site-packages (5.2.0)
```

Мы можем использовать **--no-cache-dir** вместе с **-I**, чтобы перезаписать это

```
pip install --no-cache-dir -I pillow==5.2.0
```

uninstall - удаление пакета

```
# Удаление установленного пакета
```

```
pip uninstall Flask
```

```
# Удаление пакетов перечисленных в файле
```

```
pip uninstall -r requirements.txt
```

Удаление всех установленных пакетов

```
pip freeze | xargs pip uninstall -y
```

download – закачка без установки

Позволяет скачать пакеты без установки.

```
pip3 download Flask
```

Пакеты скачиваются с зависимостями и имеют расширения .whl Установить их в проект можно через `install` так:

```
pip install --find-links=/download Flask-2.1.1-py3-none-any.whl
```

pip list

Позволяет просмотреть список всех установленных в системе пакетов.

Пример:

```
pip3 list
```

pip show

Позволяет просмотреть информацию об установленном в системе пакете.

Пример:

```
pip3 show Flask
```

В дополнение к pip show есть пакет

Virtualenv

Когда программист Python работает над большим количеством проектов, со временем у него появляется потребность в том, чтобы не устанавливать все пакеты из всех проектов себе в систему, а отделить их друг от друга. Для решения этой проблемы был создан Virtualenv.

virtualenv — программа для создания и управления окружениями Python. Позволяет создать среду со своими отдельными модулями, настройками и программами. Среда ограничивается рамками одного каталога. Очень удобна для работы с различными версиями одних и тех же модулей, для создания проектов, у которых "всё с собой", которые не зависят от операционной системы.

Установка virtualenv

Установить virtualenv можно через менеджер пакетов pip.

```
pip3 install virtualenv
```

virtualenv [OPTIONS] DEST_DIR

Обязательным параметром является только DEST_DIR - путь к папке, в которой будет хранится виртуальное окружение.

Название папки лучше называть по имени проекта.

Пример:

```
virtualenv course
```

Версия Python

Вы можете указать нужную вам версию интерпретатора python, при этом он должен быть установлен в системе. Если вы опустили эту опцию, то будет использоваться умолчательный (Выполните which python чтобы узнать какой он у вас, но скорее всего это будет /usr/bin/python).

Пример:

```
virtualenv --python=python3.6 course
```

site-packages

Запретить использование системного site-packages (для полной изоляции вашего окружения от системы). Например у вас в системе установлена "Flask 2.1", если вы будете использовать эту опцию, то в созданном окружении эта "Flask" не будет доступна.

Пример:

```
virtualenv --no-site-packages course
```

--system-site-packages

Эта опция противоположна предыдущей, то есть заставляет окружение использовать установленные в системе пакеты, если не нашлись онные в окружении.

Пример:

```
virtualenv --system-site-packages course
```

--clear

Используется для очистки существующего окружения от пакетов и прочих изменений.

Пример:

```
virtualenv --clear existing_venv
```

Использование virtualenv

После создания виртуальное окружение надо запустить, иначе будет использоваться ваше системное окружение для Python

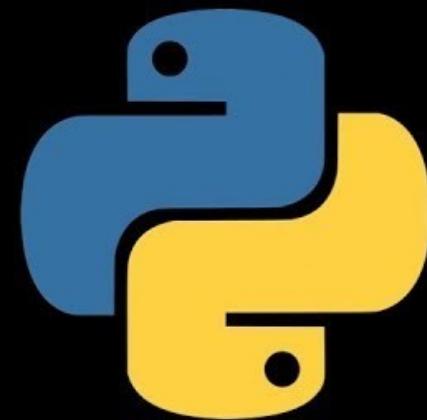
```
source venv/bin/activate
```

Когда активируется виртуальное окружение, строка приглашения к вводу в консоле (терминале) заменится на название виртуального окружения . После этого можно свободно пользоваться виртуальной средой, запускать модули python и пользоваться пакетами, установленными в виртуальном окружении. Чтобы выйти из виртуальной среды, нужно ввести:

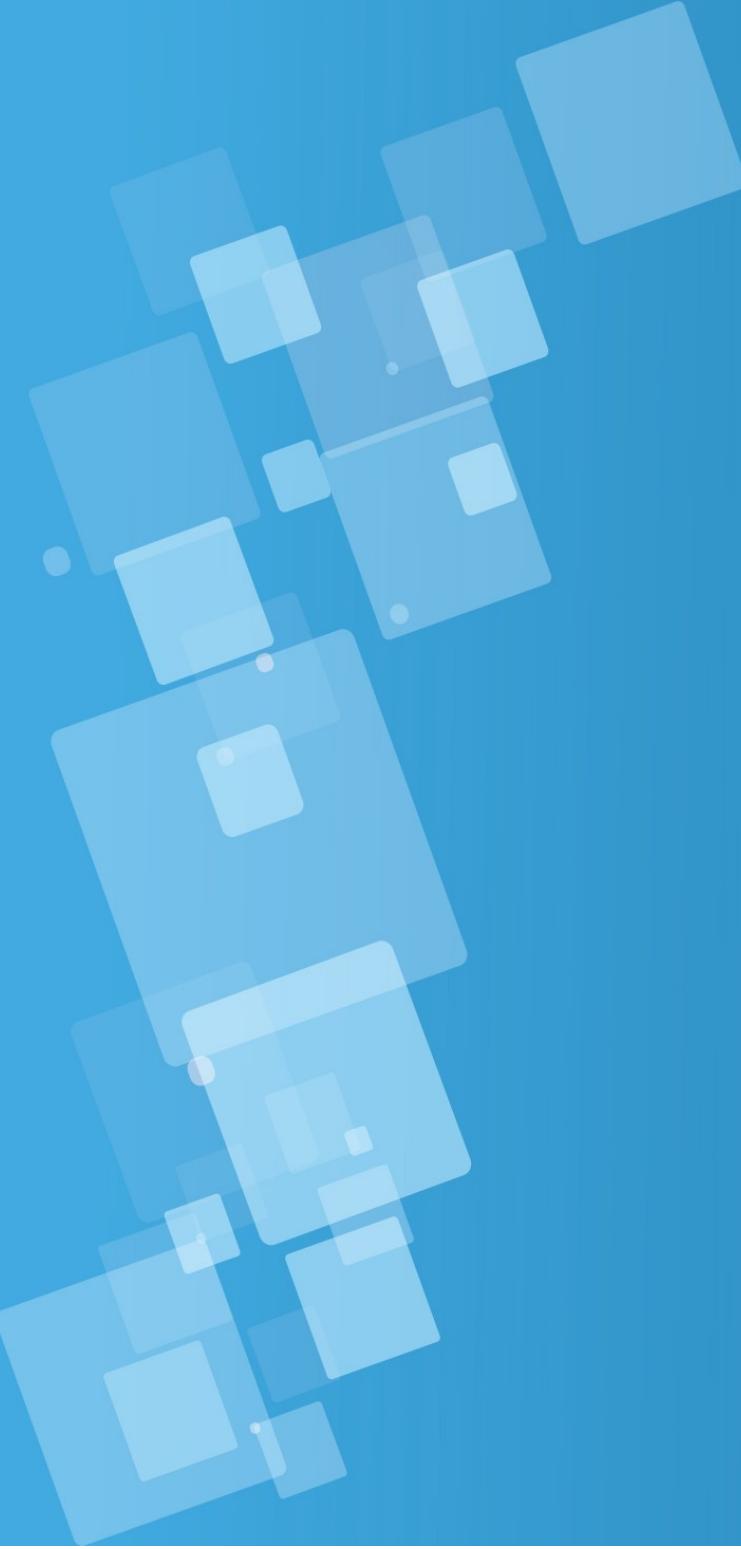
```
deactivate
```



Спасибо за внимание!



PYTHON PROGRAMMING



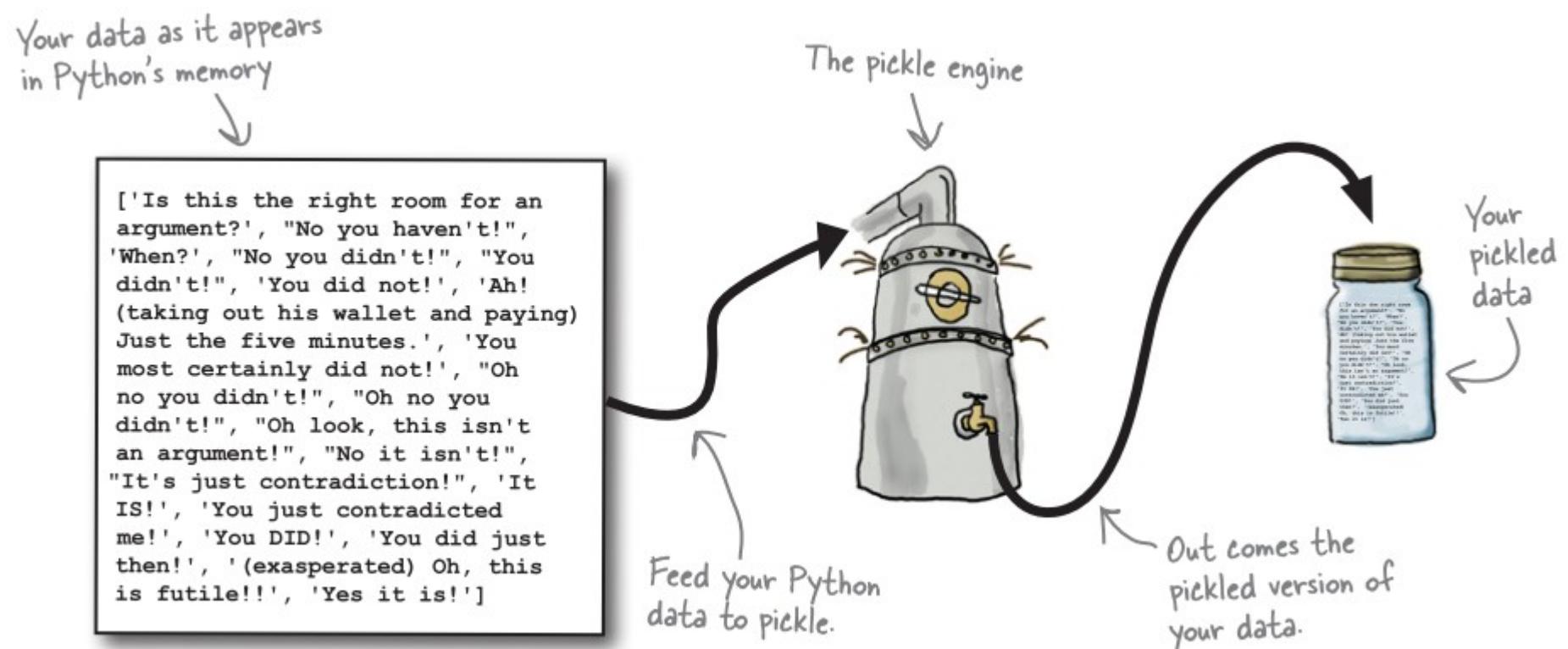
Сериализация

десериализация



PYTHON SERIALIZATION WITH PICKLE

Pickle - алгоритм сериализации и десериализации объектов Python.



Сериализация/десериализация

Сериализация (англ. **serialization**) – процесс перевода какой-либо структуры данных в любой другой, более удобный для хранения формат. Обратной к операции сериализации является операция десериализации (англ. **deserialization**) – восстановление начального состояния структуры данных из битовой последовательности.

Самой основной структурой данных в языке программирования Python является объект. Сериализация и десериализация объектов используется в том случае, если нам надо передавать информацию между запусками одной программы или между несколькими программами.



Способы сохранить/восстановить объект.

1. Pickle
2. JSON
3. YAML

pickle

Модуль `pickle` реализует мощный алгоритм сериализации и десериализации объектов Python. "Pickling" - процесс преобразования объекта Python в поток байтов, а "unpickling" - обратная операция, в результате которой поток байтов преобразуется обратно в Python-объект. Так как поток байтов легко можно записать в файл, модуль `pickle` широко применяется для сохранения и загрузки сложных объектов в Python.

Часто сериализация используется для сохранения пользовательских данных между разными сессиями работы приложения, обычно игры.

Что можно консервировать ?

- None, True или False
- Числа , вещественные и комплексные числа
- Строки , байт строки.
- Списки, наборы , картежи и словари
- Функции определенные def в глобальной области
- Классы определенные в глобальной области
- Экземпляры классов у которых можно вызвать `_dict_`

Интерфейс модуля `dir(pickle)`

```
['ADDITEMS', 'APPEND', 'APPENDS', 'BINBYTES', 'BINBYTES8', 'BINFLOAT',
'BINGET', 'BININT', 'BININT1', 'BININT2', 'BINPERSID', 'BINPUT',
'BINSTRING', 'BINUNICODE', 'BINUNICODE8', 'BUILD', 'BYTEARRAY8',
'DEFAULT_PROTOCOL', 'DICT', 'DUP', 'EMPTY_DICT', 'EMPTY_LIST',
'EMPTY_SET', 'EMPTY_TUPLE', 'EXT1', 'EXT2', 'EXT4', 'FALSE', 'FLOAT',
'FRAME', 'FROZENSET', 'FunctionType', 'GET', 'GLOBAL', 'HIGHEST_PROTOCOL',
'INST', 'INT', 'LIST', 'LONG', 'LONG1', 'LONG4', 'LONG_BINGET',
'LONG_BINPUT', 'MARK', 'MEMOIZE', 'NEWFALSE', 'NEWOBJ', 'NEWOBJ_EX',
'NEWTRUE', 'NEXT_BUFFER', 'NONE', 'OBJ', 'PERSID', 'POP', 'POP_MARK',
'PROTO', 'PUT', 'PickleBuffer', 'PickleError', 'Pickler', 'PicklingError',
'PyStringMap', 'READONLY_BUFFER', 'REDUCE', 'SETITEM', 'SETITEMS',
'SHORT_BINBYTES', 'SHORT_BINSTRING', 'SHORT_BINUNICODE', 'STACK_GLOBAL',
'STOP', 'STRING', 'TRUE', 'TUPLE', 'TUPLE1', 'TUPLE2', 'TUPLE3',
'UNICODE', 'Unpickler', 'UnpicklingError', '_Framer',
'_HAVE_PICKLE_BUFFER', '_Pickler', '_Stop', '_Unframer', '_Unpickler',
'_all__', '__builtins__', '__cached__', '__doc__', '__file__',
 '__loader__', '__name__', '__package__', '__spec__', '_compat_pickle',
 '_dump', '_dumps', '_extension_cache', '_extension_registry',
 '_getattribute', '_inverted_registry', '_load', '_loads', '_test',
 '_tuple_size2code', 'bytes_types', 'codecs', 'compatible_formats',
 'decode_long', 'dispatch_table', 'dump', 'dumps', 'encode_long',
 'format_version', 'io', 'islice', 'load', 'loads', 'maxsize', 'pack',
 'partial', 're', 'sys', 'unpack', 'whichmodule']
```

Методы модуля

Модуль pickle предоставляет следующие методы, чтобы сделать процесс pickling (пиклинг) более удобным:

Синтаксис:

```
pickle.dump(obj, file, protocol=None, *,  
fix_imports=True, buffer_callback=None)
```

Записывает сериализованный объект в файл. Дополнительный аргумент protocol указывает используемый протокол. По умолчанию равен 3 и именно он рекомендован для использования в Python 3 (несмотря на то, что в Python 3.4 добавили протокол версии 4 с некоторыми оптимизациями). В любом случае, записывать и загружать надо с одним и тем же протоколом.

```
import pickle  
  
obj = {"one": 123, "two": [1, 2, 3]}  
fd = open("data.pkl", "wb")  
pickle.dump(obj, fd, pickle.HIGHEST_PROTOCOL)
```

pickle.dumps

Возвращает обработанное представление объекта obj в виде байтового объекта, без записи его в файл.

Синтаксис:

```
pickle.dumps(obj, protocol=None, *, fix_imports=True,  
buffer_callback=None)
```

Параметры:

obj - объект Python, подлежащий сериализации,

file - файловый объект

protocol=None - протокол сериализации

fix_imports=True - сопоставление данных Python2 и Python3

buffer_callback=None - сериализация буфера в файл как часть потока pickle.

Пример:

```
import pickle  
obj = {"one": 123, "two": [1, 2, 3]}  
output = pickle.dumps(obj, 2)
```

pickle.load

Восстанавливает сериализованный объект из файла

Синтаксис:

```
pickle.load(file, *, fix_imports=True,  
encoding="ASCII", errors="strict")
```

Версия протокола определяется автоматически

Считайте байтовое представление объекта из открытого файла и возвращает сериализованный объект.

Пример:

```
import pickle  
  
with open("data.pkl", "rb") as f:  
    obj = pickle.load(f)
```

`pickle.loads`

Восстанавливает сериализованный объект в обычное представление из байтового.

Синтаксис:

```
pickle.loads(bytes_object, *, fix_imports=True,  
encoding="ASCII", errors="strict")
```

```
import pickle  
  
obj = {"one": 123, "two": [1, 2, 3]}  
  
output = pickle.dumps(obj)  
  
new_obj = pickle.loads(output)
```



JSON Serialization



Пользоваться pickle лучше только в рамках python-приложения. При обмене данными между разными языками программирования обычно используют модуль json. Также pickle никак не решает вопрос безопасности данных. Поэтому не следует десериализовать данные из неизвестных источников.

Для работы с форматом JSON в Python используется модуль json

```
import json
```

Интерфейс модуля dir(json)

```
[ 'JSONDecodeError', 'JSONDecoder', 'JSONEncoder',
'__all__', '__author__', '__builtins__', '__cached__',
['__doc__', '__file__', '__loader__', '__name__',
'__package__', '__path__', '__spec__', '__version__',
'__default_decoder', '__default_encoder', 'codecs',
'decoder', 'detect_encoding', ''dump'', ''dumps'',
'encoder', 'load', 'loads', 'scanner']
```

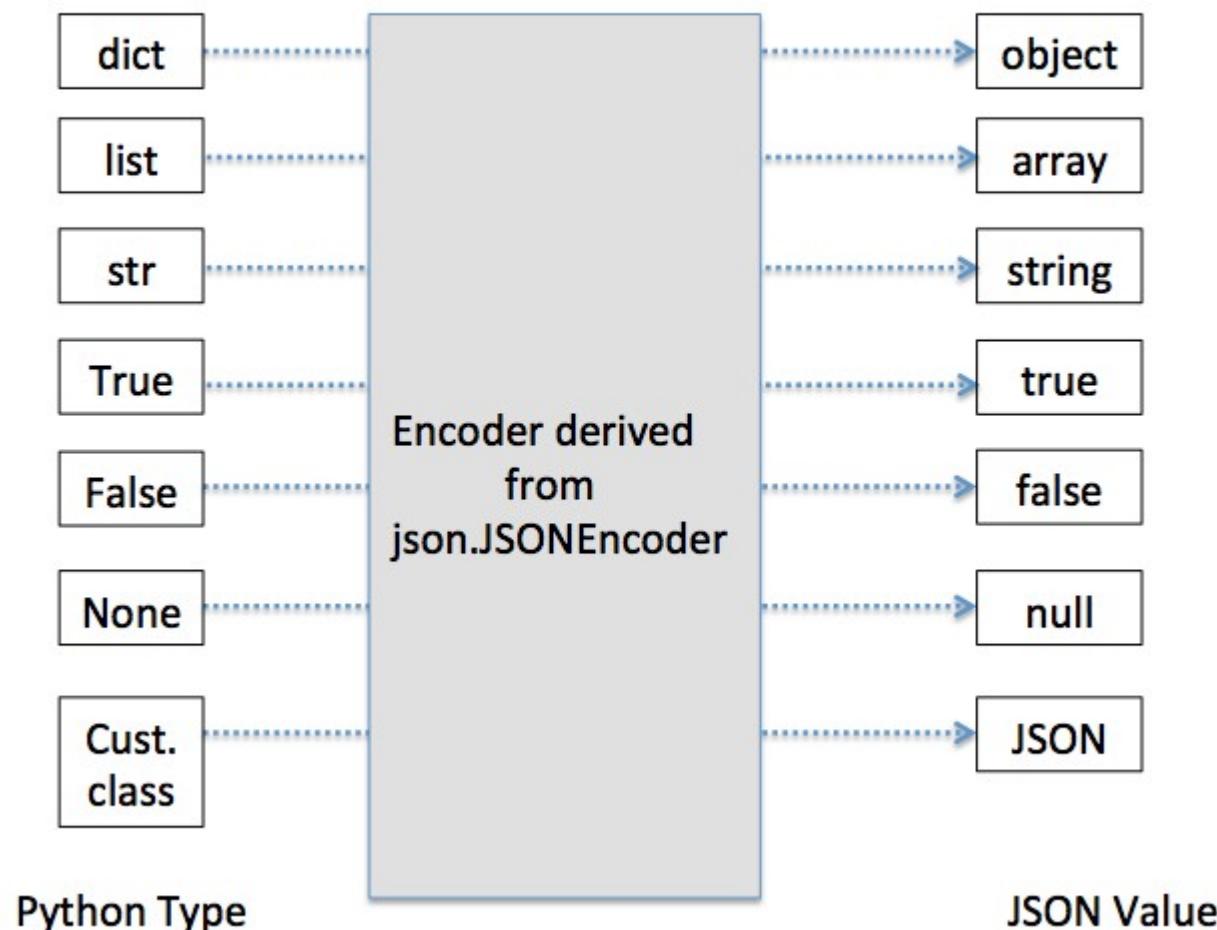
Сериализация и десериализация

Сериализация JSON

Модуль `json` предоставляет удобный метод `dump()` для записи данных в файл. Существует также метод `dumps()` для записи данных в обычную строку. Типы данных Python кодируются в формат JSON в соответствии с интуитивно понятными правилами преобразования, представленными в виде таблице ниже.

Кодировщики и декодировщики

Serialization using specialized classes of `json.JSONEncoder`



Сериализация и десериализация в строку

json.dumps(object) – сериализует obj в строку JSON-формата

json.loads(output) – десериализует строку содержащую документ JSON в объект Python.

```
import json  
obj = {"one": 123, "two": [1, 2, 3]}
```

```
output = json.dumps(obj)  
new_obj = json.loads(output)
```

Во что превратится tupl – ?

Сериализация в файл

json.dump(obj, ff, indent="") – сериализует объект в файл.

```
import json  
obj = {"one": 123, "two": [1, 2, 3]}  
  
with open("data.json", "wt") as f:  
    json.dump(obj, f, indent=4)
```

indent – количество отступов при записи

Десериализация из файла

json.load(file_descriptor) – десериализует содержимое файла

```
import json  
with open("data.json", "rt") as f:  
    obj = json.load(f)
```

Десериализация из файла

```
import json

with open("data.json", "rt") as f:
    obj = json.load(f)
```



YAML Serialization



YAML — это язык для сериализации данных, который отличается простым синтаксисом и позволяет хранить сложноорганизованные данные в компактном и читаемом формате.

Для работы с форматом YAML в Python используется модуль `yaml`

Установка

```
$ pip install ruyaml
```

Подключение

```
>>> import yaml
```

Интерфейс модуля dir(yaml)

```
[ 'StreamEndToken', 'StreamStartEvent', 'StreamStartToken',
'TagToken', 'Token', 'UnsafeLoader', 'ValueToken',
'YAMLError', 'YAMLObject', 'YAMLObjectMetaclass',
'__builtins__', '__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__', '__path__',
['__spec__'], '__version__', '__with_libyaml__', '_yaml',
'add_constructor', 'add_implicit_resolver',
'add_multi_constructor', 'add_multi_representer',
'add_path_resolver', 'add_representer', 'compose',
'compose_all', 'composer', 'constructor', 'cyaml', 'dump',
'dump_all', 'dumper', 'emit', 'emitter', 'error', 'events',
'full_load', 'full_load_all', 'io', 'load', 'load_all',
'loader', 'nodes', 'parse', 'parser', 'reader',
'representer', 'resolver', 'safe_dump', 'safe_dump_all',
'safe_load', 'safe_load_all', 'scan', 'scanner',
'serialize', 'serialize_all', 'serializer', 'tokens',
'unsafe_load', 'unsafe_load_all', 'warnings']
```

Сериализация и десериализация

yaml.dump(obj) – сериализация объекта в строку

`yaml.load(new_obj)` – десериализация объекта в строку

```
import yaml
```

```
from yaml.loader import SafeLoader
```

```
obj = {"one": 123, "two": [1, 2, 3]}
```

```
output = yaml.dump(obj)
```

```
new_obj = yaml.load(output, Loader=SafeLoader)
```

Сериализация в файл

```
import yaml

obj = {"one": 123, "two": [1, 2, 3]}
with open("data.yml", "wt") as f:
    yaml.dump(obj, f)
```

Десериализация из файла

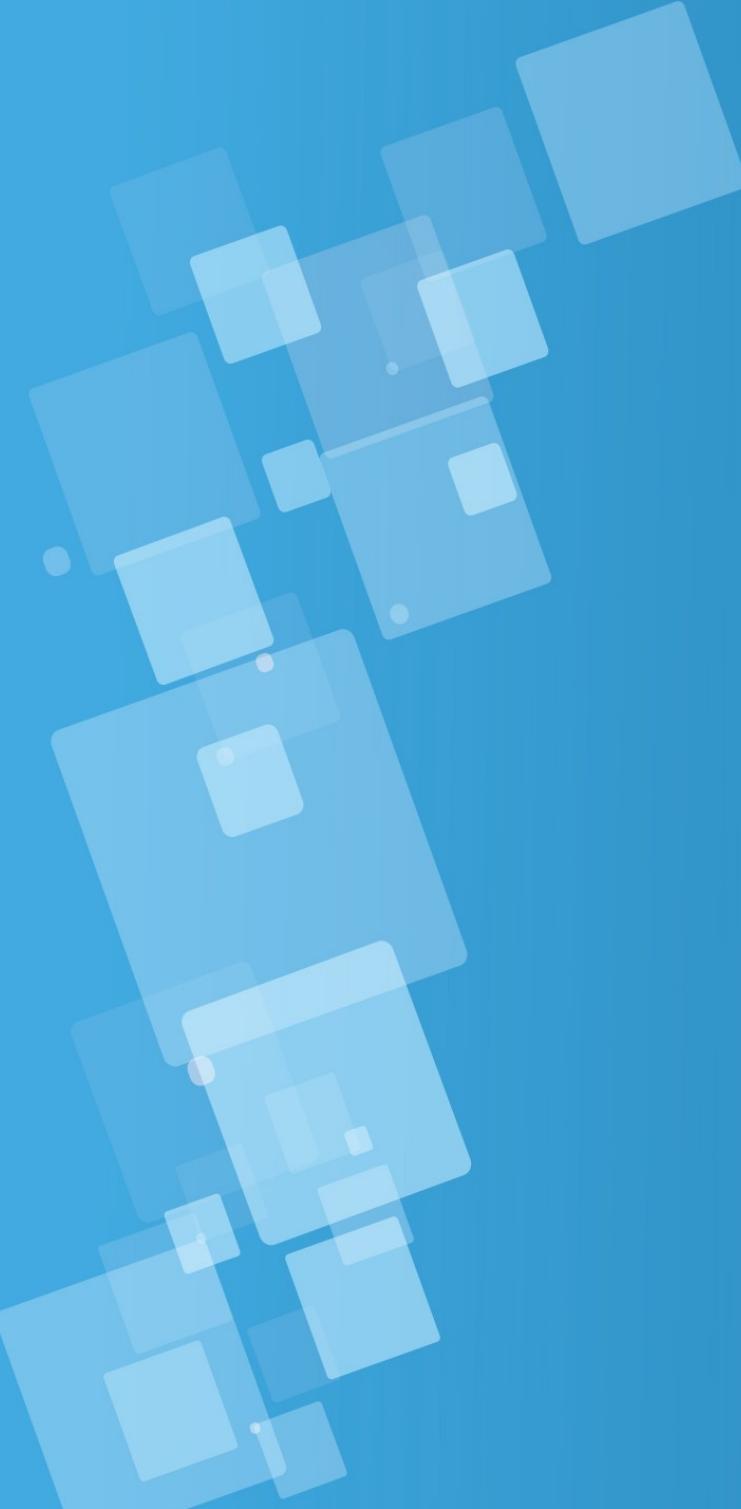
```
import yaml

with open("data.yml", "wt") as f:
    obj = yaml.load(f)
```

Заключение

Сериализация и десериализация объектов Python является важным аспектом распределенных систем. Вам часто приходится взаимодействовать с другими системами, реализованными на других языках, а иногда просто нужно сохранить состояние своей программы в постоянном хранилище.

Python поставляется с некоторыми схемами сериализации в своей стандартной библиотеке, а многие другие доступны в качестве сторонних модулей.



ФУНКЦИИ

Часть II

Лямбда-функции, анонимные функции

Раньше мы использовали функции, обязательно связывая их с каким-то именем. В Python есть возможность создания односторонних анонимных функций

Конструкция:

```
lambda [param1, param2, ..]: [выражение]
```

lambda – функция, возвращает свое значение в том месте, в котором вы его объявляете.

Лямбда-функция

```
# функция, которая возвращает свой параметр  
def identity(x):  
    return x  
  
identity(100)  
  
#identity() функция тождества принимает передаваемый  
аргумент в x и возвращает его при вызове.
```

Лямбда-функция :

lambda x: x

Ключевое слово: lambda

Параметр: x

Выражение (тело) : x

Вызов lambda

Для вызова lambda обернем функцию и ее аргумент в круглые скобки. Передадим функции аргумент.

```
>>> (lambda x: x + 1) (2)
```

3

Именование **lambda**

Поскольку лямбда-функция является выражением, оно может быть именовано. Поэтому вы можете написать предыдущий код следующим образом:

```
>>> add_one = lambda x: x + 1
```

```
>>> add_one(2)
```

3

Аргументы

Как и обычный объект функции, определенный с помощью `def`, лямбда поддерживают все различные способы передачи аргументов. Это включает:

- Позиционные аргументы
- Именованные аргументы (иногда называемые ключевыми аргументами)
- Переменный список аргументов (часто называемый `*args`)
- Переменный список аргументов ключевых слов `*kwargs`

Аргументы функции

Функции с несколькими аргументами (функции, которые принимают более одного аргумента) выражаются в лямбда-выражениях Python, перечисляя аргументы и разделяя их запятой (,), но не заключая их в круглые скобки:

```
>>> full_name = lambda first, last: f'Full name:  
{first.title()} {last.title()}'  
  
>>> full_name('guido', 'van rossum')  
'Full name: Guido Van Rossum'
```

Именованные параметры

Как и в случае **def**, для аргументов **lambda** можно указывать стандартные значения.

```
>>>str = ( lambda a='He', b='ll' , c='o': a+b+c)  
str(a='Ze')  
'Zello'
```

Пример

```
>>> (lambda x, y, z: x + y + z)(1, 2, 3)
6
>>> (lambda x, y, z=3: x + y + z)(1, 2)
6
>>> (lambda x, y, z=3: x + y + z)(1, y=2)
6
>>> (lambda *args: sum(args))(1, 2, 3)
6
>>> (lambda **kwargs: sum(kwargs.values()))(one=1,
two=2, three=3)
6
>>> (lambda x, *, y=0, z=0: x + y + z)(1, y=2, z=3)
```

ФУНКЦИИ ВЫСОКОГО ПОРЯДКА

Лямбда-функция может быть функцией более высокого порядка, принимая функцию (нормальную или лямбда-функцию) в качестве аргумента, как в следующем надуманном примере:

```
>>> high_ord_func = lambda x, func: x + func(x)
```

```
>>> high_ord_func(2, lambda x: x * x)
```

6

```
>>> high_ord_func(2, lambda x: x + 3)
```

7



Для чего используется lambda ?

Использование лямбда-выражения как литерала списка.

```
import random

# Словарь, в котором формируются три случайные числа
# с помощью лямбда-выражения

L = [ lambda : random.random(),
      lambda : random.random(),
      lambda : random.random() ]

# Вывести результат

for l in L:

    print(l())
```

Лямбда-выражения как литералы кортежей

Формируется кортеж, в котором элементы умножаются на разные числа.

```
import random
# Кортеж, в котором формируются три литерала-строки
# с помощью лямбда-выражения
T = ( lambda x: x*2,
      lambda x: x*3,
      lambda x: x*4 )
# Вывести результат для строки 'abc'
for t in T:
    print(t('abc'))
```

Результат:

```
abcabc
abcabcaabc
abcabcaabcabc
```

Использование лямбда-выражения для формирования таблиц переходов.

```
# Словарь, который есть таблицей переходов  
Dict = {
```

```
    1 : (lambda: print('Monday')) ,  
    2 : (lambda: print('Tuesday')) ,  
    3 : (lambda: print('Wednesday')) ,  
    4 : (lambda: print('Thursday')) ,  
    5 : (lambda: print('Friday')) ,  
    6 : (lambda: print('Saturday')) ,  
    7 : (lambda: print('Sunday'))  
}
```

```
# Вызывать лямбда-выражение, выводящее название  
вторника
```

```
Dict[2]() # Tuesday
```

Таблица переходов , в которой вычисляется площадь известных фигур.

```
import math

Area = {
    'Circle' : (lambda r: math.pi*r*r), # окружность
    'Rectangle' : (lambda a, b: a*b), # прямоугольник
    'Trapezoid' : (lambda a, b, h: (a+b)*h/2.0) # трапеция
}

# Вызвать лямбда-выражение, которое выводит площадь окружности
# радиуса 2
print('Area of circle = ', Area['Circle'](2))

# Вывести площадь прямоугольника размером 10*13
print('Area of rectangle = ', Area['Rectangle'](10, 13))

# Вывести площадь трапеции для a=7, b=5, h=3
areaTrap = Area['Trapezoid'](7, 5, 3)
print('Area of trapezoid = ', areaTrap)
```

Совместное использование lambda-функции со встроенными функциями

Функция **filter()** принимает два параметра – функцию и список для обработки. В примере мы применим функцию `list()`, чтобы преобразовать объект `filter` в список.

Пример 1

```
numbers=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
list(filter(lambda x:x%3==0, numbers))
```

```
[0, 3, 6, 9]
```

Код берет список `numbers`, и отфильтровывает все элементы из него, которые не делятся нацело на 3. При этом фильтрация никак не изменяет изначальный список.

filter()

Пример 2

```
even = lambda x: x%2 == 0  
list(filter(even, range(11)))  
[0, 2, 4, 6, 8, 10]
```

Обратите внимание, что filter() возвращает итератор, поэтому необходимо вызывать list, который создает список с заданным итератором.

Реализация, использующая конструкцию генератора списка, дает одинаковый результат:

```
>>> [x for x in range(11) if x%2 == 0]  
[0, 2, 4, 6, 8, 10]
```

Функция **map()**

Функция **map()** в отличие от функции **filter()** возвращает значение выражения для каждого элемента в списке.

#Пример 1

```
numbers=[0,1,2,3,4,5,6,7,8,9,10]
```

```
list(map(lambda x:x%3==0, numbers))
```

```
[True, False, False, True, False, False, True, False,  
False, True, False]
```

#Пример 2

```
list(map(lambda x: x.capitalize(), ['cat', 'dog',  
'cow']))
```

```
['Cat', 'Dog', 'Cow']
```

reduce()

ФИО

функция `reduce()` принимает два параметра — функцию и список. Сперва она применяет стоящую первым аргументом функцию для двух начальных элементов списка, а затем использует в качестве аргументов этой функции полученное значение вместе со следующим элементом списка и так до тех пор, пока весь список не будет пройден, а итоговое значение не будет возвращено. Для того, чтобы использовать `reduce()`, вы должны сначала импортировать ее из модуля `functools`.

reduce()

Пример

```
from functools import reduce  
numbers=[0,1,2,3,4,5,6,7,8,9,10]  
reduce(lambda x,y:y-x,numbers)  
5
```

$$1 - 0 = 1$$

$$2 - 1 = 1$$

$$3 - 1 = 2$$

$$4 - 2 = 2$$

$$5 - 2 = 3$$

$$6 - 3 = 3$$

$$7 - 3 = 4$$

$$8 - 4 = 4$$

$$9 - 4 = 5$$

$$10 - 5 = 5$$



Можно ли в лямбда-выражениях
использовать стандартные операторы
управления **if**, **for**, **while**?

НЕТ

НО !

Можно использовать тернарный оператор.

```
lower = (lambda x, y: x if x < y else y)
```

```
# Вызов 1 способ
```

```
(lambda x, y: x if x < y else y)(10, 3)
```

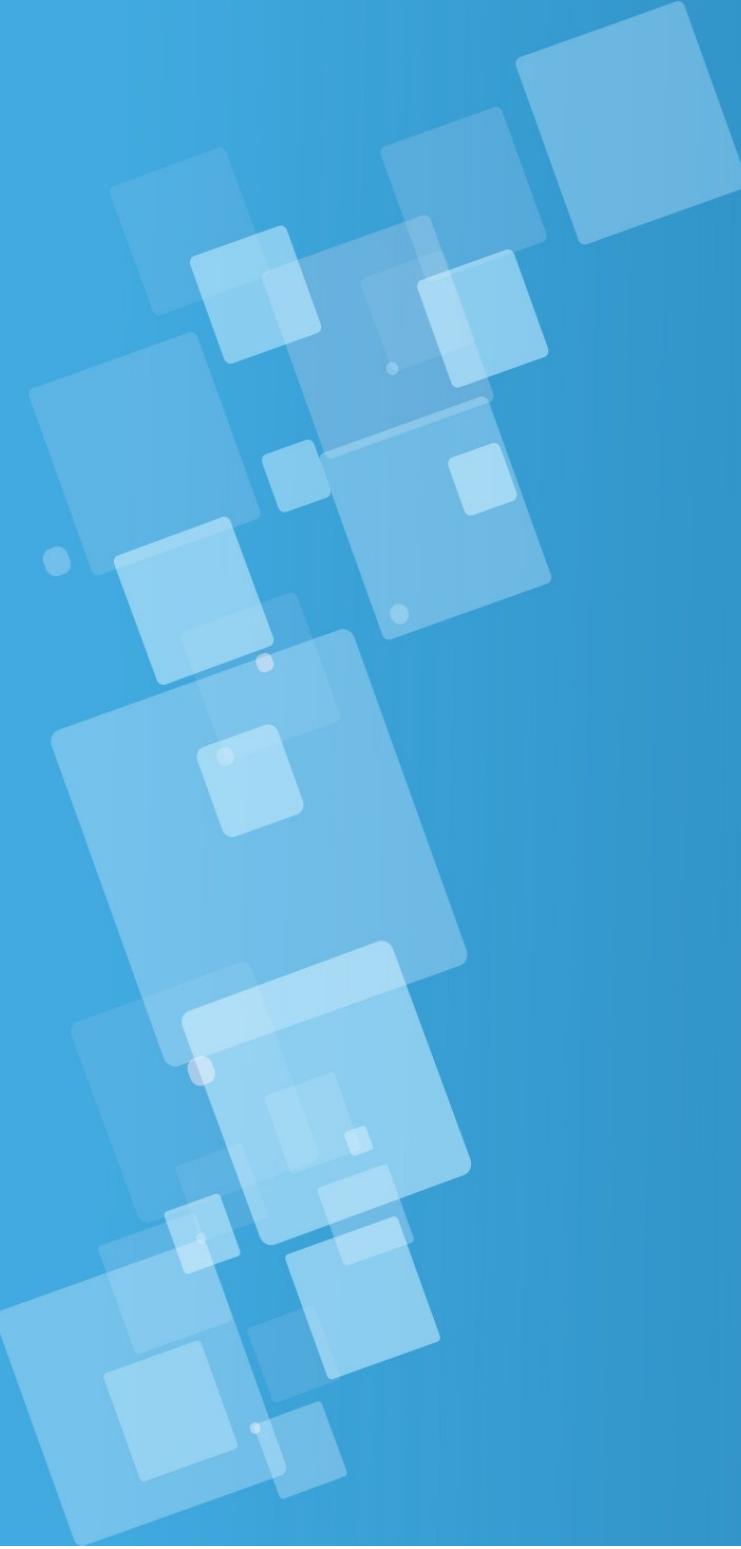
```
# Вызов 2 способ
```

```
lower(10, 3)
```

3

Заключение

- Избегать чрезмерного использования лямбд
- Использовать лямбды с функциями высшего порядка или ключевыми функциями Python
- Функции более высокого порядка, такие как map(), filter() и functools.reduce(), могут быть преобразованы в более элегантные формы с небольшими изменениями, в частности, со списком или генератором выражений.



ФУНКЦИИ

Часть III
Декораторы

ФУНКЦИИ – ЭТО ОБЪЕКТЫ

Функции Python относятся к объектам. Их можно **присваивать переменным**, хранить в структурах данных (коллекциях), **передавать их в качестве аргументов** другим функциям и даже **возвращать** их в качестве значений из других функций.



Почему функция является в языке
Python объектом ?

Пример присвоения переменной

Поскольку функция render является объектом. Ее можно присвоить еще одной переменной, точно также как это происходит с любым другим объектом.

```
def render(text):
    print(text.upper() + '!')

render('Hello')
#Hello!

show = render
show('Wellcome')
#Wellcome!

print(show is render)                      # True
del render
print(show.__name__)
#render
```

Передача функций в качестве аргумента

Поскольку функции являются объектами, их можно передавать в качестве аргументов другим функциям.

```
def render(text):  
    return text.upper() + ' !'
```

```
def call_hello(func):      ← функция более выс. пор.  
    result = func('Hello')  
    print(result)
```

```
>>> call_hello(render)
```

Функция более высокого порядка

Функции которые принимают в качестве аргументов другие функции называют функциями более высокого порядка.

Классический пример таких функций это встроенная функция **map**, которая принимает в качестве аргументов: объект функцию и итерируемый объект. А затем вызывает эту функцию с каждым элементом итерируемого объекта, выдавая результат по мере прохождения итерируемого объекта.

```
def up_cap(text):  
    return text.capitalize()  
  
lst = list(map(up_cap, ['cat', 'dog', 'cow']))  
print(lst)
```

ФУНКЦИИ МОГУТ БЫТЬ ВЛОЖЕННЫМИ

Python допускает определение функций внутри других функций. Такие функции называются вложенными функциями (nested function) или внутренними функциями (inner function)

```
def speak(text):  
    def whisper(t):  
        return t.lower() + '....'  
    return whisper(text)  
>>>print(speak('Hello, World'))
```

Всякий раз, когда вы вызываете функцию `speak`, она определяет новую функцию `whisper` и затем после этого ее вызывает

Вложенная функция

Внимание! Вложенная функция `whisper` не существует за пределами функции `speak`

```
def speak(text):  
  
    def whisper(t):  
  
        return t.lower() + '...'  
  
    return whisper(text)
```

```
# Попытка вызвать функцию whisper  
>>> whisper('Hello, World')  
  
NameError: name 'whisper' is not defined
```

Вопрос! Как получить доступ к вложенной функции `whisper` за пределами функции `speak`?

Возврат функции в качестве значения

Не забывайте функции являются объектами – и вы можете вернуть вложенную функцию в качестве значения.

```
def speak():
    def whisper(t):
        return t.lower() + '...'
    return whisper

# Получаем из функции speak объект функции whisper
wr = speak()

# Вызываем функцию с одним аргументом
print(wr('Hello, World'))
```

ФУНКЦИИ МОГУТ ЗАХВАТЫВАТЬ ЛОКАЛЬНЫЕ СОСТОЯНИЯ

Перепишем функцию speak следующим образом

```
def speak(text):  
    def whisper():  
        print(text.lower() + '...')  
    return whisper  
  
foo = speak('Hello World')  
bar = speak('Wellcome')  
foo() → hello, world...  
bar() → wellcome...
```

Внутренние функции получают доступ к родительскому параметру `text`, определенному в родительской функции. Такой доступ называется лексическим замыканием или для краткости замыканием. Замыкание помнит значения из своего лексического контекста, даже когда поток управления программы больше не находится в этом контексте.

Ключевые выводы

- В Python абсолютно все является объектом, включая функции. Их можно присваивать переменным, передавать в функции более высокого порядка а также возвращать из них.
- Функции могут быть вложенными , и они могут захватывать и уносить с собой часть состояния родительской функции. Функции которые это делают, называются замыканиями.



Python

Function Decorator

Декаратор функции.

СВИДЕТЕЛЬ

Alisa: *Декораторы – они что украсят нашу функцию ?*

Bob: *Нет , декоратор обертывает другую функцию и позволяет исполнять программный код до и после того, как обернутая функция выполниться.*

Декаратор функции.

Alisa: А что нужно чтобы написать декоратор ?

Bob: Объяви функцию декоратор , в нее передай функцию которую будешь обертывать. Реализуй в декораторе вложенную функцию - обертку (*wrapper*) в котором будет содержаться логика до или после выполнения передаваемой функции. Вызови во вложенной функции , функцию из параметра. Верни вложенную функцию из декоратора.

Декоратор

```
def decorator_render(func):    ← функция декоратор
    print("Call decorator function")
    def wrapper(text):          ← вложенная функция
        print(f"Логика перед вызовом функции")
        func(text)
        print("Логика после вызова функции")
    return wrapper

def render(text):
    print(f"Это функция render выводит {text.upper() }")

# Вызываем декоратор и передаем туда функцию
>>> wrap_func = decorator_render(render)

#Получаем из декоратора вложенную функ. и вызываем её
>>> wrap_func('аргумент1')
```

Аргументы для вложенной функции

```
def decorator_render(func):
    print("Call decorator function")
    def wrapper(*args, **kwargs):
        print(f"Логика перед вызовом функции")
        func(*args, **kwargs)
        print("Логика после вызова функции")
    return wrapper

def render(text):
    print(f"Это функция render выводит {text.upper() }")

wrap_func = decorator_render(render)
wrap_func('аргумент1')
```

Оператор @

Вызов декоратора можно переписать так:

```
def decorator_render(func):  
    print("Call decorator function")  
    def wrapper(text):  
        print(f"Логика перед вызовом функции")  
        res = func(text)  
        print("Логика после вызова функции")  
        return res  
    return wrapper
```

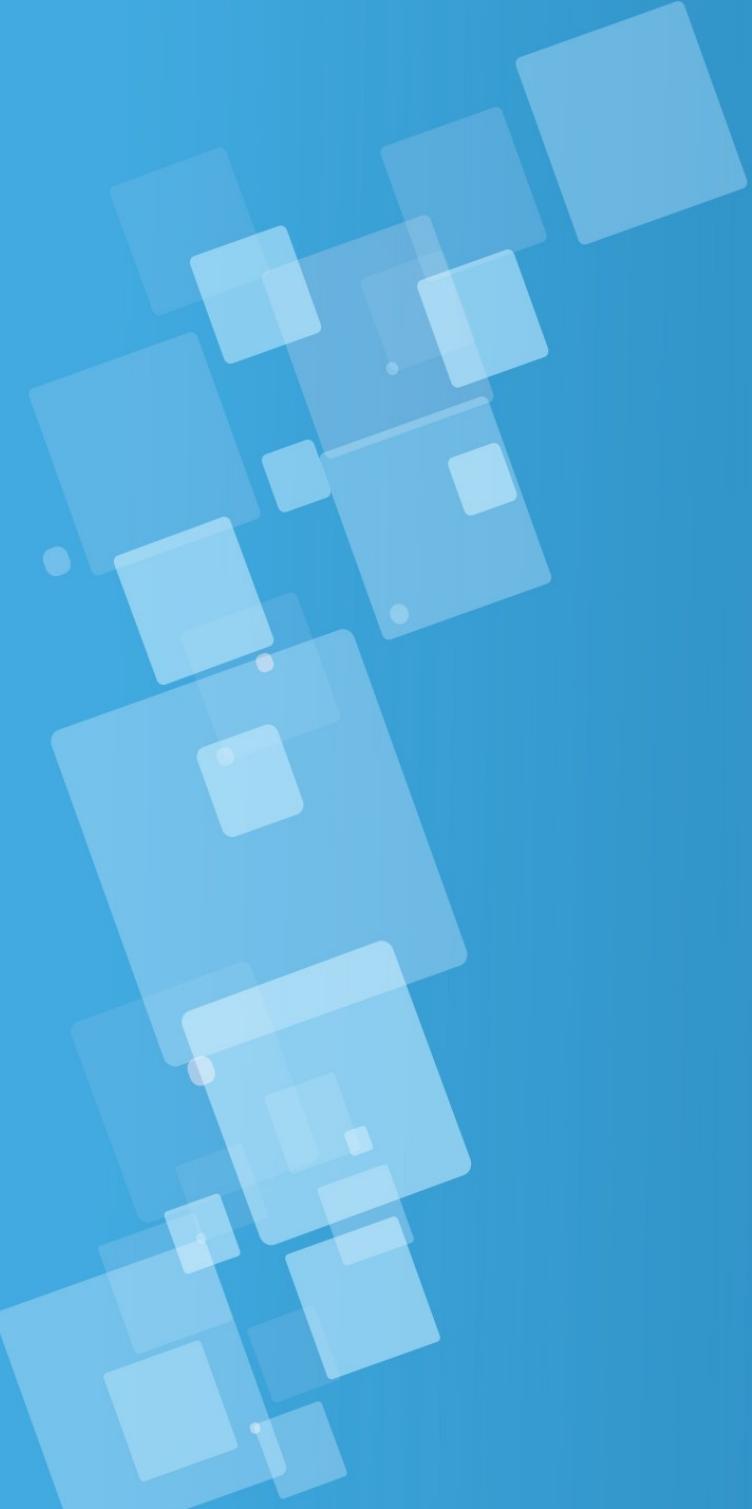
```
@decorator_render
```

```
def render(text):  
    return f"Это функция render выводит {text.upper()}"  
render('аргумент1')
```



Для чего использовать декораторы?

- Ведение протокола операции (журналирование)
- Обеспечение контроля за доступом и аутентификацией
- Функции хронометража
- Ограничение частоты вызова API
- Кеширование и др.



PEP 289 – Generator Expressions

Python List Comprehension

```
[expression(x) for x  
in existing_list if  
condition(x)]
```



Списковые включения

Списковые включения или генераторы списков – это способ построения нового списка за счет применения **выражения** к каждому элементу в последовательности, который связан с циклом **for** а также инструкции **if-else** для определения того, что в итоге окажется в финальном списке.

Пример

Do this	For this collection	In this situation
[x**2 for x in range(0, 50) if x % 3 == 0]		

Способы формирования списков

- 1) при помощи циклов
- 2) при помощи функции map()
- 3) при помощи list comprehension

1. При помощи цикла for

```
s = []
for i in range(10):
    s.append(i ** 3) # Добавляем к списку куб каждого
# числа
print(s)
# [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

2. При помощи функции map()

```
list(map(lambda x: x ** 3, range(0,10)))
# [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Лаконично!

3. При помощи конструкции **list comprehension**

```
[x**3 for x in range(10)]
# [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Условие в конце включения

[«возв. значение» for «элемент списка» in «список» **if** «условие»]

#Получить все нечетные цифры в диапазоне от 0 до 9

```
[x for x in range(10) if x%2 == 1]
```

```
# [1, 3, 5, 7, 9]
```

Возвведение в квадрат

```
[x**2 for x in range(10)]
```

Условие в начале включения

```
# Замена отрицательного диапазона нулем  
  
>>> original_prices = [1.25, -9.45, 10.22, 3.78, -  
5.92, 1.16]  
  
>>> prices = [i if i > 0 else 0 for i in  
original_prices]  
  
>>> prices  
[1.25, 0, 10.22, 3.78, 0, 1.16]
```

Условие в начале включения

```
from string import ascii_letters

letters = 'һытҔтгҔзҔп' # набор букв из разных
алфавитов

# Разграничиваем буквы на английские и не английские

is_eng = [f'{letter}-ДА' if letter in ascii_letters
else f'{letter}-НЕТ' for letter in letters]

# ['h-ДА', 'ы-НЕТ', 't-ДА', 'Ӯ-НЕТ', 'т-НЕТ', 'r-ДА',
'ц-НЕТ', 'з-НЕТ', 'q-ДА', 'п-НЕТ']
```

Вызов функции в выражении генераторов

```
# Замена отрицательного диапазона нулем  
  
original_prices = [1.25, -9.45, 10.22, 3.78, -5.92,  
1.16]  
def get_price(price):  
    return price if price > 0 else 0  
  
prices = [get_price(i) for i in original_prices]
```

Вложенная генерация

Представим список из слов, который мы хотим привести к сплошному списку из букв. Двойная итерация: по словам и по буквам

```
words = ['я', 'изучаю', 'Python']
```

```
res = [letter for word in words for letter in word]  
print(letters)
```

```
>>>res  
['я', 'и', 'з', 'у', 'ч', 'а', 'ю', 'Р', 'у', 'т', 'h', 'о', 'н']
```

Вложенная генерация

```
key = ["name", "age", "weight"]
```

```
value = ["Lilu", 25, 100 ]
```

```
[{x, y} for x in key for y in value ]
```

```
[
```

```
{'Lilu', 'name'}, {25, 'name'}, {100, 'name'},
```

```
{'Lilu', 'age'}, {25, 'age'}, {100, 'age'},
```

```
{'weight', 'Lilu'}, {'weight', 25}, {'weight', 100}
```

```
]
```

Вложенная генерация

```
>>> matrix = [[i for i in range(5)] for _ in range(6)]
>>> matrix
[
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4],
    [0, 1, 2, 3, 4]
]
```

Внешний генератор [... for _ in range(6)] создает 6 строк в то время как внутренний генератор [i for i in range(5)] заполняет каждую строку значениями.

Вложенная генерация

```
# Преобразование матрицы в плоский вид
```

```
matrix = [
...     [0, 0, 0],
...     [1, 1, 1],
...     [2, 2, 2],
...
]
>>> flat = [col for row in matrix for col in row]
>>> flat
[0, 0, 0, 1, 1, 1, 2, 2, 2]
```

Вложенная генерация

```
# Генерация таблицы умножения от 1 до 5  
  
T = [ [x*y for x in range(1, 6)] for y in range(1, 6) ]  
  
print(T)  
[[1, 2, 3, 4, 5],  
 [2, 4, 6, 8, 10],  
 [3, 6, 9, 12, 15],  
 [4, 8, 12, 16, 20],  
 [5, 10, 15, 20, 25]]
```

Когда использовать генератор списков ?

- Использовать для выполнения простых фильтраций, модификаций или форматирования итерируемых объектов.
- Для увеличение производительности.
- Для компактности
- Следует избегать использования генератора списков, если вам нужно добавить слишком много условий это делает код трудным для чтения.



Python

Dictionary Comprehension

Генераторы словарей

Генерация словаря похожа на генерацию списка и предназначена для создания словаря.

```
d = {}  
for num in range(1, 10):  
    d[num] = num**2  
print(d)  
{1:1, 2:4, 3:9, 4:16, 5:25, 6:36, 7:49, 8:64, 9: 81}
```

```
D = { num: num**2 for num in range(1, 10) }  
>>>d  
{1:1, 2:4, 3:9, 4:16, 5:25, 6:36, 7:49, 8:64, 9: 81}
```

Генераторы словарей

#Создадим словарь по списку кортежей

```
items = [ ('c', 3), ('d', 4), ('a', 1), ('b', 2) ]
```

```
dict_variable = { key:value for (key,value) in items }
```

```
print(dict_variable)
```

Что если не будет значения :value ?

Set comprehensions!

Условие if

```
# Добавим в конструкцию генератора условие фильтрации  
  
dict1 = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}  
# Проверка, больше ли элемент, чем 2  
  
filtered = {k:v for (k,v) in dict1.items() if v>2}  
  
print(filtered)  
# {'e': 5, 'c': 3, 'd': 4}
```

Условие if

Фильтрация по возрасту

```
ages = {  
    'kevin': 12,  
    'marcus': 9,  
    'evan': 31,  
    'nik': 31  
}  
f = {k:val for (k, val) in ages.items() if val > 25}  
print(new_ages)
```

Несколько условий if

#Последовательные операторы if работают так, как если бы между ними были логические **and**.

```
dict = {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
r = {k:v for (k,v) in dict.items() if v>2 if v%2 == 0}
print(r)
# {'d': 4}
```

Вложенные генераторы словарей

```
dict = {'first': {'a': 1}, 'second': {'b': 2}}  
fd = {o_key: {float(i_val) for (i_key, i_val) in o_val.items()}  
      for (o_key, o_val) in dict.items()}  
  
print(fd)  
# {'first': {1.0}, 'second': {2.0}}
```

Код имеет вложенный генератор словаря, то есть один генератор внутри другого. Как видите, вложенный генератор словаря может быть довольно трудным как для чтения, так и для понимания. Использование генераторов при этом теряет смысл (ведь мы их применяем для улучшения читабельности кода).

Использование enumerate функции

```
names = ['Harry', 'Hermione', 'Ron', 'Neville', 'Luna']  
index = {k:v for (k, v) in enumerate(names)}  
print(index)  
{'Harry':0, 'Hermione':1, 'Ron':2, 'Neville':3, 'Luna':4}
```



Когда использовать генераторы словарей?

Во всех случаях что и при генерации списков.

Заключение

Подобные конструкции позволяют создавать не только списки (list comprehension) и словари (dictionary comprehension), генераторы (generator expression – при помощи «()»), а также множества (set comprehension – при помощи «{ }» и картежи «tuple()»). Принцип везде один и тот же.