

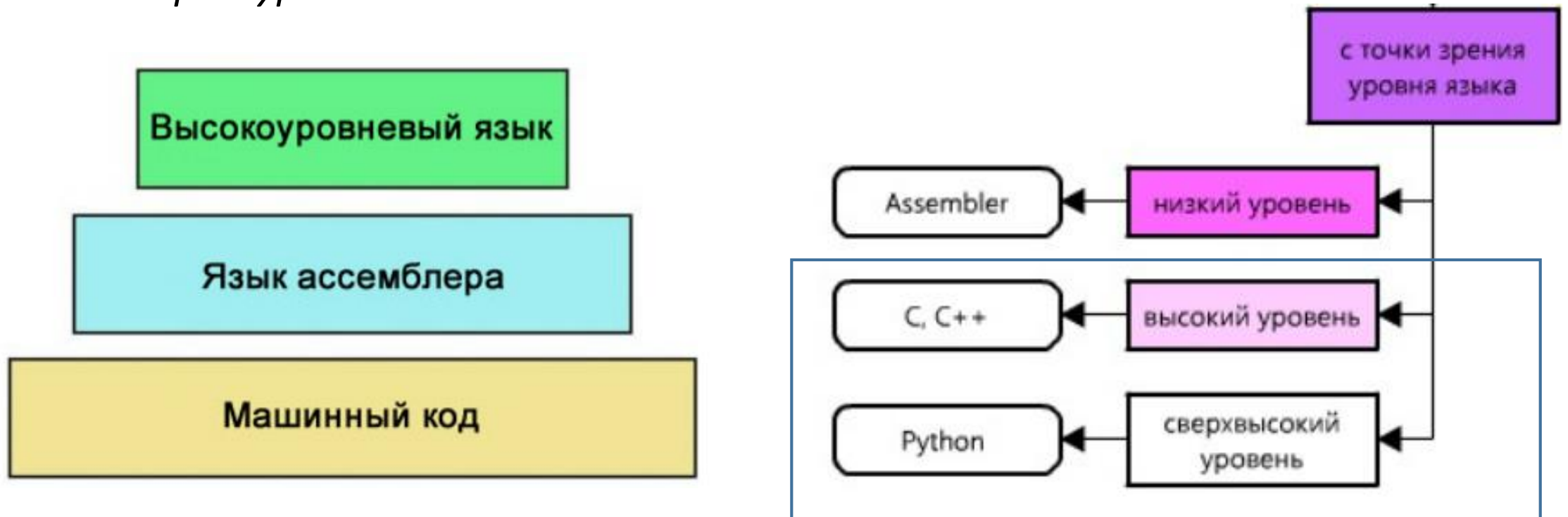
Высокоуровневые языки программирования

Лектор: Сухорукова Ирина Геннадьевна
ст. преподаватель кафедры программной инженерии
ауд.408 к.1
контакт в телеграмме – у старост групп

Языки программирования классифицируются на различные уровни в зависимости от их близости к машинному коду и аппаратному уровню.

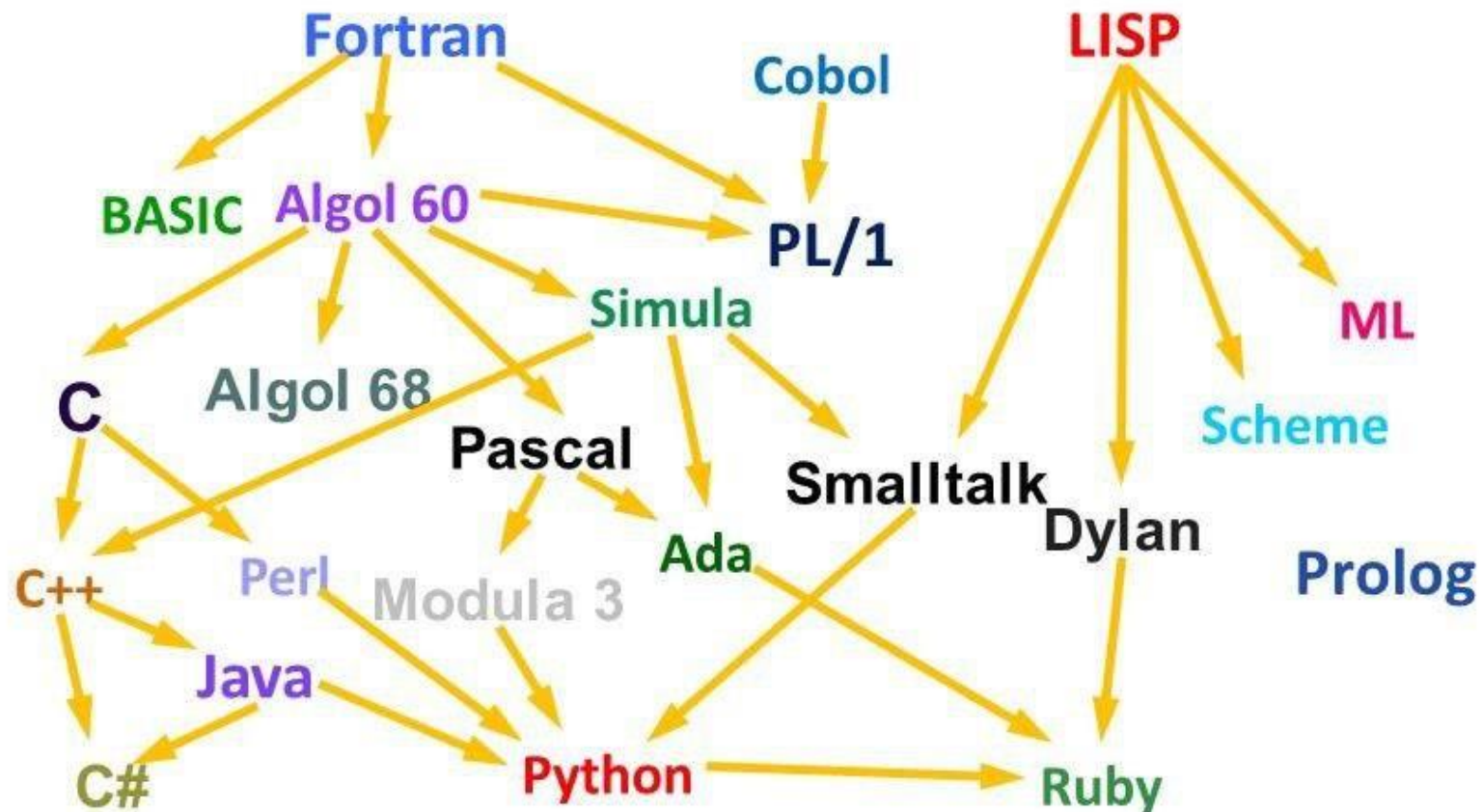
Высокоуровневые языки программирования обеспечивают более абстрактный и удобный синтаксис, позволяющий разработчикам писать программы на более высоком уровне абстракции.

Низкоуровневые языки программирования предоставляют больший контроль над аппаратурой и машинным кодом.



A family tree of languages

Some of the 2400 + programming languages



Отличия высокоуровневых от низкоуровневых ЯП

Абстракция. Высокоуровневые языки предоставляют более высокий уровень абстракции, что позволяет разработчикам сосредоточиться на решении задачи, не заботясь о деталях аппаратного уровня.

Память. Чем "ниже" язык, тем чаще нужно будет работать с памятью компьютера, самостоятельно производить очистки, заполнения и так далее. Высокоуровневые языки берут большинство этих задач на себя, вам остаётся просто писать код. Однако, из-за этого будет тратиться больше ресурсов системы.

Портируемость. Чем "ниже" язык, тем он менее портируем, так как его набор инструкций более "машинозависим" и предназначен для конкретных платформ. Программы, написанные на высокоуровневых ЯП, платформонезависимы, так как использование трансляторов и интерпретаторов обеспечивает их связь с различными ОС и оборудованием.

Скорость. Тут низкоуровневые языки в выигрыше, так как они взаимодействуют напрямую с процессором и регистрами памяти, а не с интерпретаторами и компиляторами. Если говорить проще, то на высоких языках программы пишутся на человекоподобном языке и компьютеру нужно время, чтобы перевести ваше письмо в удобный для себя вид.





Простота использования. Высокоуровневые языки больше адаптированы к человеческому языку и поэтому на них проще писать.

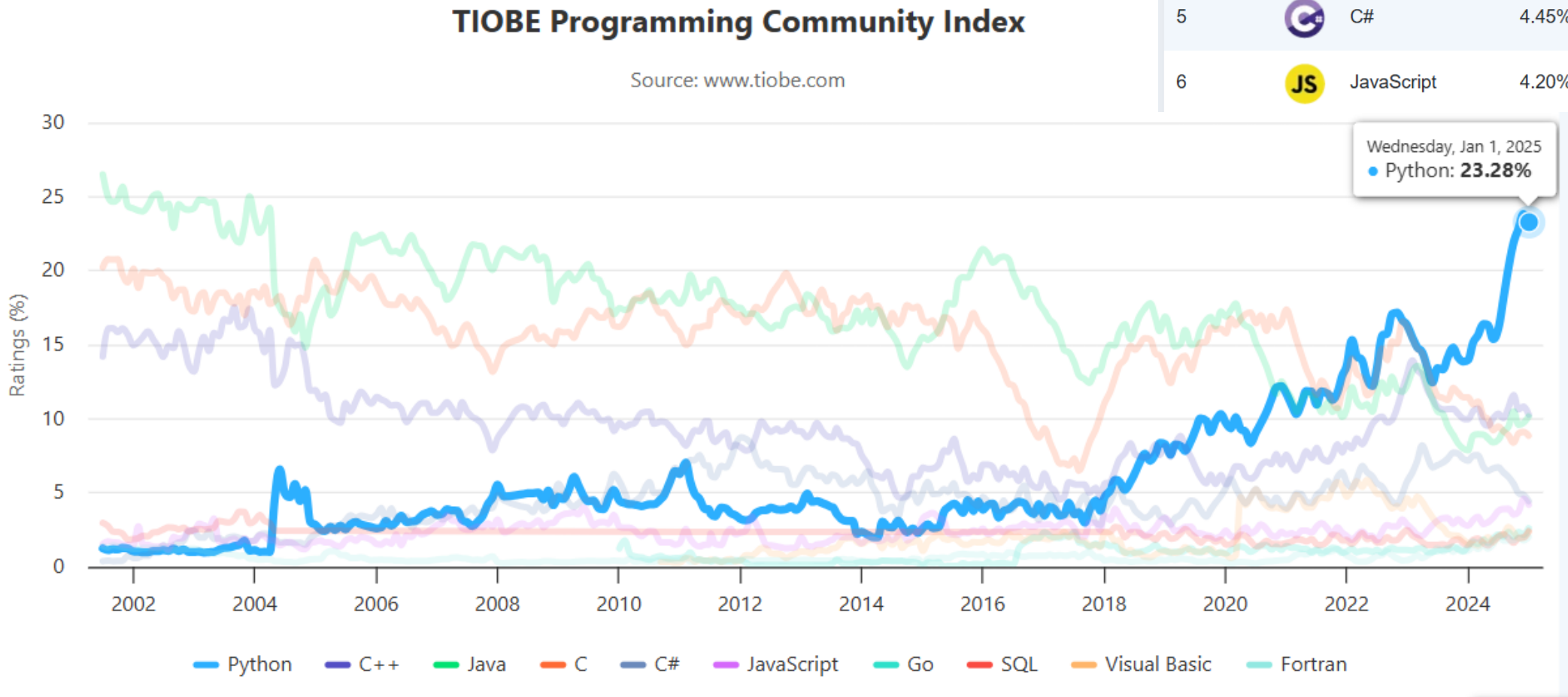
Практически все ЯП, которые широко используются сейчас — высокоуровневые. Давайте рассмотрим первую десятку такого списка:

1	Python	Машинное обучение, тестирование, боты, сайты, скрипты
2	C	Сложные задачи, взаимодействие с «железом»
3	C++	Программное обеспечение, операционные системы, драйвера
4	Java	Широкое применение
5	C#	Приложения, базы данных, машинное обучение, игры
6	Visual Basic	Приложения для Windows и других платформ
7	JavaScript	Скрипты, интернет
8	SQL	Базы данных
9	Язык ассемблера	Драйверы, вирусы-антивирусы, взаимодействие с «железом»
10	PHP	Сайты, интернет

Индекс TIOBE — индекс, оценивающий популярность языков программирования, на основе подсчёта результатов поисковых запросов, содержащих название языка.

<https://www.tiobe.com/tiobe-index/>

Jan 2025	Programming Language	Ratings	Change
1	 Python	23.28%	+9.32%
2	 C++	10.29%	+0.33%
3	 Java	10.15%	+2.28%
4	 C	8.86%	-2.59%
5	 C#	4.45%	-2.71%
6	 JavaScript	4.20%	+1.43%





Питон или Пайтон?

Язык Python назван в честь телешоу комик-группы Монти **Пайтон**.

Python невероятно эффективен: программы, написанные на нем, делают больше, чем многие на других языках и в меньшем объеме кода.

Его просто учить

Главной целью основателя *Python*, Гвидо ван Россума, было создать простой и понятный широкому кругу людей язык программирования. Изучение любого языка требует усидчивости и дисциплины. Но *Python* в этом смысле считается одним из самых комфортных, особенно для новичков. Простой синтаксис позволяет легко учиться и читать код.

Он очень распространенный

Python универсален благодаря богатой стандартной библиотеке, поэтому его применяют в самых разных областях:

- веб-разработке;
- machine Learning и AI;
- Big Data;
- разработке игр.



Логотип, использовавшийся с 1990-х до 2006 года



Гвидо Ван Россум

PEP 20 – Дзен Python или «Дзен Питона» — это набор принципов и рекомендаций, которые отражают философию и ценности языка программирования Python. Эти принципы помогают разработчикам писать чистый, понятный и эффективный код. Философия Python основана на простоте, ясности и удобочитаемости кода. Вот несколько ключевых принципов «Дзен Питона»:

1. **Красивое лучше, чем уродливое.** Код должен быть написан так, чтобы его было приятно читать и понимать. Это помогает другим разработчикам легче разбираться в вашем коде.
2. **Явное лучше, чем неявное.** Лучше использовать простые и понятные конструкции, чем сложные и запутанные. Это делает код более надёжным и лёгким для понимания.
3. **Простое лучше, чем сложное.** Не стоит усложнять код, если можно сделать его проще. Это помогает избежать ошибок и упрощает поддержку кода.
4. **Сложное лучше, чем запутанное.** Если код становится сложным, лучше структурировать его так, чтобы он был более понятным. Это помогает другим разработчикам легче разобраться в вашем коде.
5. **Плоское лучше, чем вложенное.** Лучше избегать глубоких вложенных структур, так как они могут сделать код трудным для понимания. Вместо этого используйте более плоские структуры.
6. **Разрешённое лучше, чем запрещённое.** Если что-то разрешено, это значит, что вы можете это делать. Если что-то запрещено, это значит, что вы не должны это делать.
7. **Ошибки никогда не должны замалчиваться.** Если в коде есть ошибка, лучше, чтобы она была обнаружена и исправлена. Это помогает сделать код более надёжным.
8. **Если не замалчивать ошибки, то лучше делать это явно.** Если вы скрываете ошибку, лучше сделать это явно, а не неявно. Это помогает другим разработчикам понять, что вы сделали и почему.
9. **Должен существовать один — и желательно только один — очевидный способ сделать это.** Если есть несколько способов сделать что-то, лучше выбрать один и придерживаться его. Это помогает сделать код более согласованным и понятным.


```
>>> '1'+'1'
'11'
>>> 1+1
2
```

В зависимости от типа
объекта операции
применяются по-разному

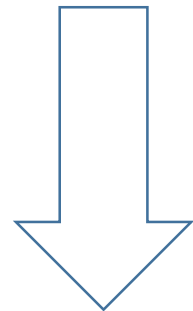
В Python все данные является
объектами.
Объект хранится в
выделанной области памяти.

```
>>> id('hello')
1508633896624
>>> id("hello")
1508633896624
>>> id('world')
1508633897008
```

```
print()  
type()  
dir()  
id()  
len()
```

Независимые функции, которые не принадлежат ни одному классу

Методы это тоже функции, но они принадлежат определенному объекту и вне его не существуют.
Посмотреть методы объекта можно функцией `dir()`



dir() позволяет узнать методы класса

```
>>> dir('5')
['_add_', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
 '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__',
 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
 'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit',
 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper',
 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix',
 'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
 'upper', 'zfill']
```

ΔΚΤΙΒΑΙ

```
>>> dir(5)
['_abs_', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delattr__',
 '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '__floordiv__',
 '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__', '__hash__',
 '__index__', '__init__', '__init_subclass__', '__int__', '__invert__', '__le__', '__lshif',
 't', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__', '__pos',
 't', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
 '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ror__', '__rroun',
 'd', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rxor__',
 '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__',
 '__trunc__', '__xor__', 'as_integer_ratio', 'bit_count', 'bit_length', 'conjugate',
 'denominator', 'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
```

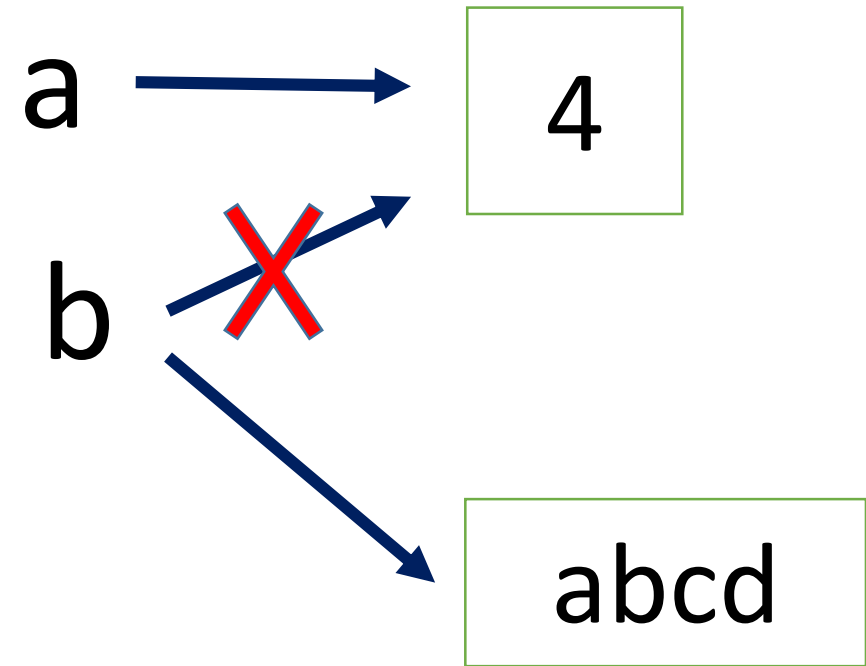
Динамическая типизация

```
>>> a=4
>>> type(a)
<class 'int'>
>>>
>>> a='hello'
>>> type(a)
<class 'str'>
```

Динамическая типизация - Python сам определяет объект какого типа объект нужно создать и затем присваивает ссылку на этот объект переменной

Объект это выделенный объем памяти который хранит указатель на тип, счетчик ссылок и ассоциированные с этим типом атрибуты и методы

```
>>> a=4
>>> id(a)
1508596580688
>>> b=a
>>> id(b)
1508596580688
>>>
>>> b='abcd'
>>> id(b)
1508633887728
```



Стандартные типы данных

- ✓ Целые числа
- ✓ Вещественные числа
- ✓ Строки
- ✓ Кортежи
- ✓ Словари
- ✓ Множества

$a, b = b, a$

PEP 8 – Руководство по стилю кода Python <https://peps.python.org/pep-0008/>

Одно из ключевых открытий Гвидо заключается в том, что код читается гораздо чаще, чем пишется. Представленные здесь рекомендации направлены на улучшение читаемости кода и обеспечение его единообразия в широком спектре кода Python. Как говорится в PEP 20, «Читаемость имеет значение».

Рекомендации к отступам при написании параметров функции

Correct:

Aligned with opening delimiter.

```
foo = long_function_name(var_one, var_two,  
                          var_three, var_four)
```

Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.

```
def long_function_name(  
    var_one, var_two, var_three,  
    var_four):  
    print(var_one)
```

Hanging indents should add a level.

```
foo = long_function_name(  
    var_one, var_two,  
    var_three, var_four)
```

Wrong:

Arguments on first line forbidden when not using vertical alignment.

```
foo = long_function_name(var one, var_two,  
                          var_three, var_four)
```

Further indentation required as indentation is not distinguishable.

```
def long_function_name(  
→ var_one, var_two, var_three,  
→ var_four):  
    print(var_one)
```


Принятые стили в Python

Функции пишутся через нижнее подчеркивание:

```
def test_guest_can_see_lesson_name_in_lesson_without_course(self, driver):
```

Классы пишут с помощью CamelCase:

```
class TestLessonNameWithoutCourseForGuest():
```

Константы пишут в стиле UPPERCASE:

```
MAIN_PAGE = "/catalog"
```

Обязательные отступы

```
a = 33
b = 200

if b > a:
→ c = b // a
→ print(c)

print("The end")
```

```
def factorial(n):
→ res = 1
→ for i in range(1, n + 1):
→     res *= i
    return res

print(factorial(3))
print(factorial(5))
```

Типы аргументов в функциях

Позиционные
аргументы
важен порядок

```
def add(a, b):  
    return a + b  
  
result = add(3, 5) # 3 и 5 – позиционные аргументы  
print(result) # Вывод: 8
```

Именованные
аргументы
*порядок не имеет
значения*

```
def greet(first_name, last_name):  
    print(f"Hello, {first_name} {last_name}!")  
  
greet(last_name="Doe", first_name="John")
```

Аргументы по
умолчанию

```
def greet(name="Guest"):  
    print(f"Hello, {name}!")  
  
greet() # Вывод: Hello, Guest!  
greet("Alice") # Вывод: Hello, Alice!
```

✓ **def** func(a, b, c=2): # c - необязательный аргумент 4 usages
 return a + b + c

func(a: 1, b: 2) # a = 1, b = 2, c = 2 (по умолчанию)

func(a: 1, b: 2, c: 3) # a = 1, b = 2, c = 3

func(a=1, b=3) # a = 1, b = 3, c = 2

func(a=3, c=6) # a = 3, c = 6, b не определен ОШИБКА!

Произвольное количество аргументов

***args** позволяет передавать произвольное количество позиционных аргументов. Это полезно, когда не известно заранее, сколько аргументов будет передано функции.

```
def add(*args):  
    return sum(args)  
  
result = add(1, 2, 3, 4)  
print(result)  # Вывод: 10
```

****kwargs** позволяет передавать произвольное количество именованных аргументов. Это полезно для создания функций с гибкими параметрами.

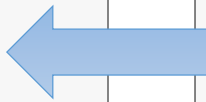
```
def print_info(**kwargs):  
    for key, value in kwargs.items():  
        print(f"{key}: {value}")  
  
print_info(name="Alice", age=30)
```

Лямбда-функция

`Lambda` аргументы: выражение

```
double = lambda x: x*2  
print(double(5))
```

```
def double(x):  
    return x * 2
```



- ✓ Лямбда-функция может иметь любое количество аргументов, но вычисляет и возвращает только одно значение.
- ✓ Лямбда-функции позволяет представить всего одно выражение.

Лямбда-функция. Пример с filter()

Функция filter() в Python принимает в качестве аргументов функцию и список .

Функция вызывается со всеми элементами в списке, и в результате возвращается новый список, содержащий элементы, для которых функция результирует в True.

Вот пример использования функции filter() для отбора четных чисел из списка.

```
my_list = [1, 3, 4, 6, 10, 11, 15, 12, 14]
new_list = list(filter(lambda x: (x%2 == 0) , my_list))
print(new_list)
```

```
[4, 6, 10, 12, 14]
```


Лямбда-функция. Пример с map()

Функция map() принимает в качестве аргументов функцию и список.

Функция вызывается со всеми элементами в списке, и в результате возвращается новый список, содержащий элементы, возвращенные данной функцией для каждого исходного элемента.

Ниже пример использования функции map() для удвоения всех элементов списка.

```
current_list = [1, 3, 4, 6, 10, 11, 15, 12, 14]
new_list = list(map(lambda x: x*2 , current_list))
print(new_list)
```

```
[2, 6, 8, 12, 20, 22, 30, 24, 28]
```

Лямбда-функция. Пример с reduce()

Функция reduce() принимает в качестве аргументов функцию и список.

Функция вызывается с помощью лямбда-функции и итерируемого объекта и возвращается новый уменьшенный результат. Так выполняется повторяющаяся операция над парами итерируемых объектов.

Функция reduce() входит в состав модуля functools.

```
from functools import reduce

current_list = [5, 15, 20, 30, 50, 55, 75, 60, 70]
summa = reduce((lambda x, y: x + y), current_list)
print(summa)
```

Функция reduce()
модуля functools
кумулятивно
применяет функцию
function к элементам
итерируемой iterable
последовательности,
сводя её к
единственному
значению.