



ТОМСКИЙ  
ПОЛИТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ



## Системный анализ процессов переработки нефти и газа

Лекция №1 - Основы  
программирования на языке Python

Вячеслав Алексеевич Чузлов  
к.т.н., доцент ОХИ ИШПР ТПУ

# Язык программирования Python

Python – высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение кроссплатформенности написанных на нём программ.

## Установка

- Установить Python (<https://www.python.org/>). В MacOS X и Linux уже установлен.
- Только для Windows. При установке выбрать опцию с заданием переменных окружения (лучше установить пакет [Anaconda](#)).

## Среда разработки

- Для обучения, работы с данными и экспериментов с кодом: Jupyter Notebook из пакета [Anaconda](#).
- Текстовые редакторы: [Visual Studio Code](#), [Sublime](#), [Atom](#), [Notepad++](#) и др.
- IDE (Integrated Development Environment): [PyCharm \(Community Edition\)](#), [Visual Studio](#), [Spyder](#) и др.

# Язык программирования Python

## Кто использует Python

- Google
- YouTube
- Dropbox
- BitTorrent
- iRobot
- Netflix
- Intel
- Cisco
- NASA

## Сильные стороны

- Качество программного кода
- Продуктивность труда
- Кроссплатформенность
- Библиотеки

Индекс ТІОВЕ (январь 2024)

Позиция	ЯП	Рейтинг
1	 Python	13.97 %
2	 C	11.44 %
3	 C++	9.96 %
4	 Java	7.87 %
5	 C#	7.16 %

# Типы данных

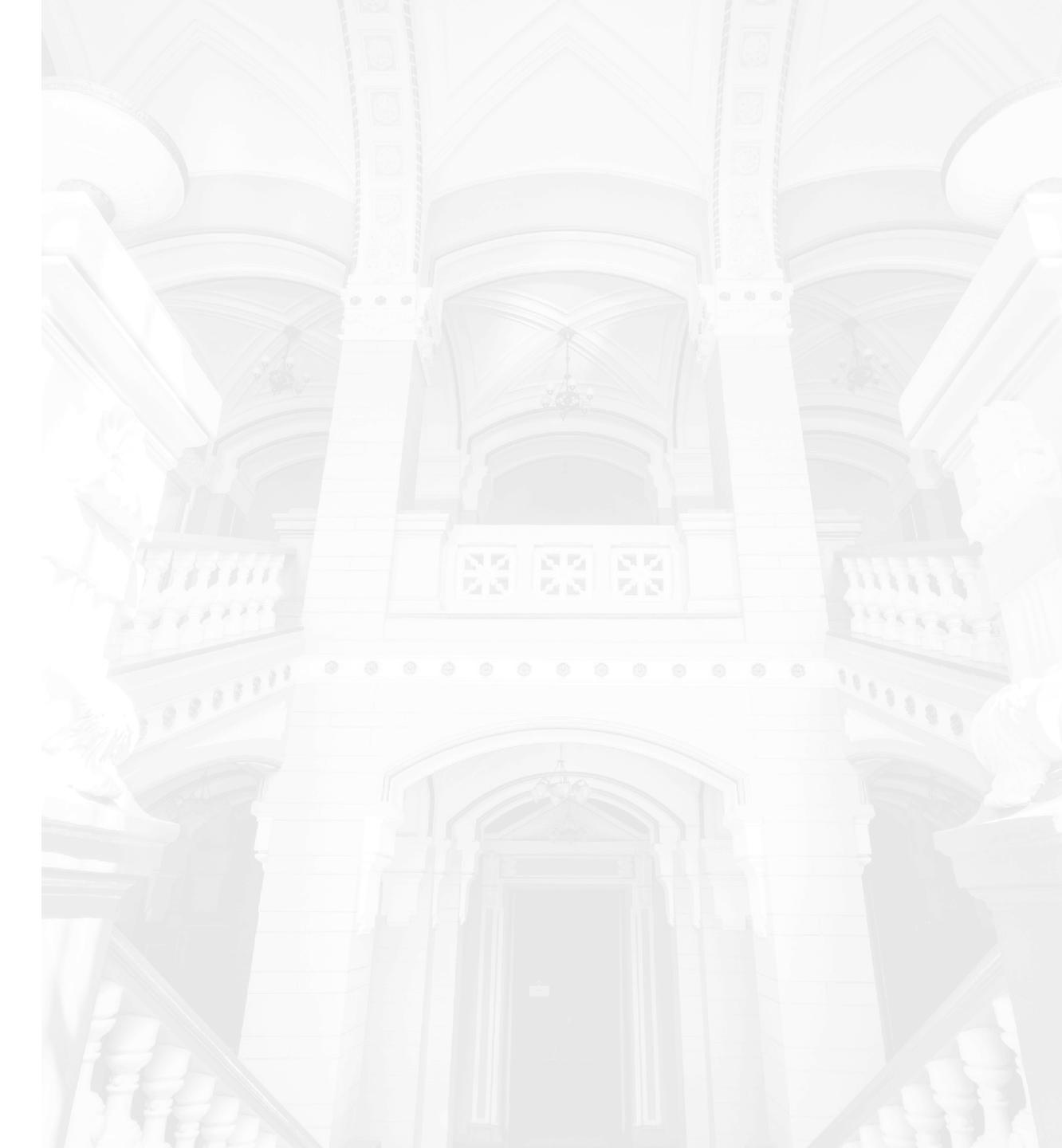


# Классификация объектов

Основные типы объектов в Python

Тип объекта	Категория	Изменяемый?
Числа ( <code>int</code> , <code>float</code> , <code>complex</code> )	Числовые	Нет
Логический ( <code>bool</code> )	Логические	Нет
Строки ( <code>str</code> )	Последовательности	Нет
Списки ( <code>list</code> )	Последовательности	Да
Словари ( <code>dict</code> )	Отображения	Да
Кортежи ( <code>tuple</code> )	Последовательности	Нет
Множества ( <code>set</code> )	Множества	Да

# Числовые типы



# Числовые типы

Числа в Python могут быть трех типов:

1. Целые числа ( `int` );
2. Числа с плавающей точкой ( `float` );
3. Комплексные числа ( `complex` ).

Литерал	Расшифровка
<code>1234</code> , <code>-24</code> , <code>0</code> , <code>9999999999999999</code>	Целые числа (неограниченный размер)
<code>1.23</code> , <code>1.</code> , <code>3.17e-10</code> , <code>4E210</code> , <code>4.0e+210</code>	Числа с плавающей точкой
<code>3+4j</code> , <code>3.0+4.0j</code> , <code>3J</code>	Литералы комплексных чисел

# Арифметические операции с числами

Основные арифметические операции с числами:

Приоритет	Операция	Описание	Тип операции
1	<code>x ** y</code>	Возведение в степень	Бинарная
2	<code>-x , +x</code>	Противоположность, идентичность	Унарная
3	<code>x / y</code>	Настоящее деление	Бинарная
3	<code>x // y</code>	Деление нацело	Бинарная
3	<code>x % y</code>	Остаток от деления (деление по модулю)	Бинарная
3	<code>x * y</code>	Умножение	Бинарная
4	<code>x - y</code>	Вычитание	Бинарная
4	<code>x + y</code>	Сложение	Бинарная

# Оператор присваивания

Для выполнения операции присваивания в Python используется оператор `=`:

```
x = 10  
y = 20  
print(x + y) # 30
```

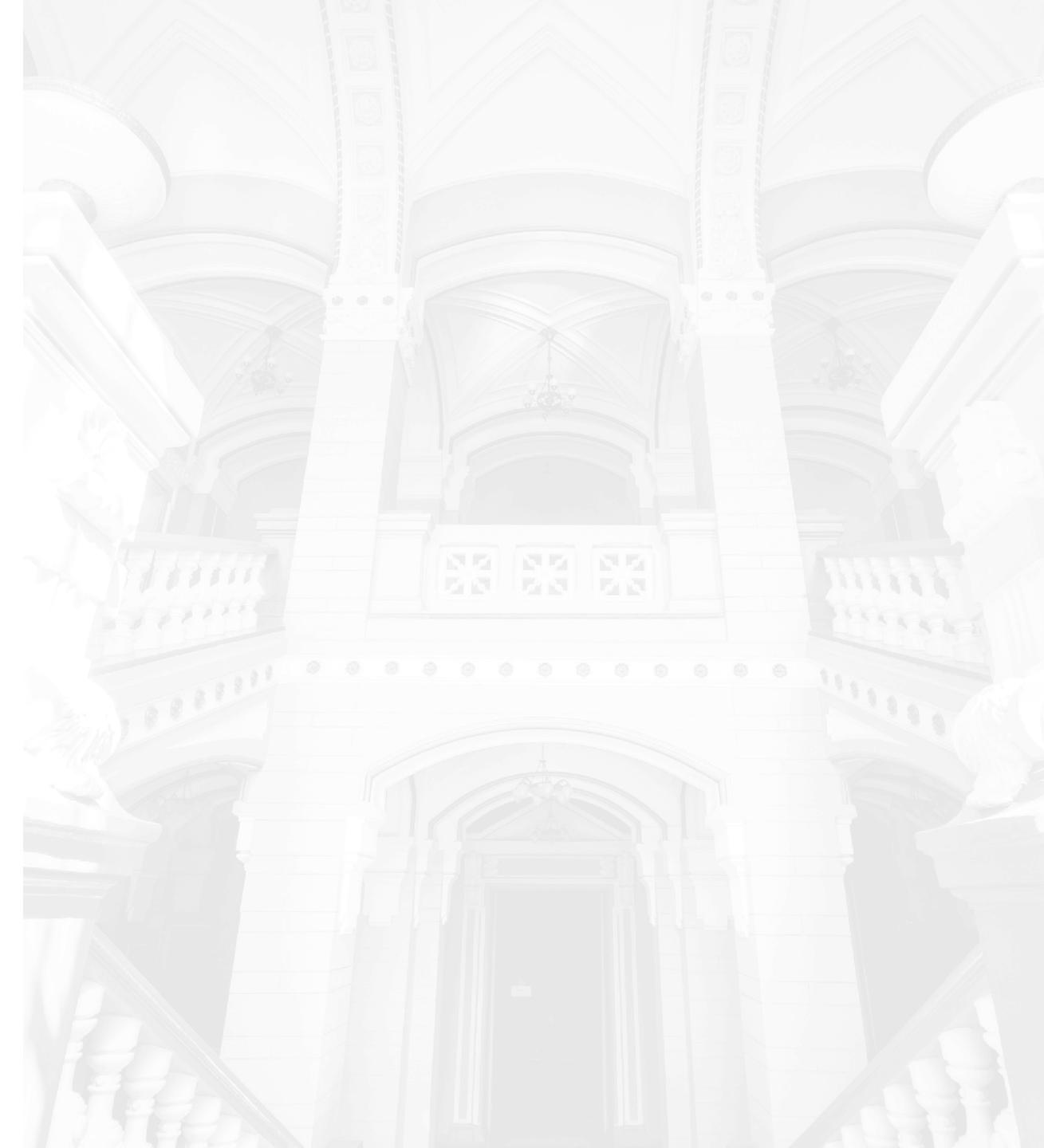
**Составные операторы присваивания** - набор операторов, сочетающих в себе бинарную операцию и оператор присваивания.

```
x = 1  
x = x + 10 # Традиционное присваивание  
print(x) # 11  
x += 10 # Составное присваивание  
print(x) # 21
```

Составные операторы присваивания:

`x += y` `x -= y` `x /= y` `x //= y` `x %= y` `x *= y` `x **= y`

# Переменные (variables)



# Что такое переменная

- При создании объекта, например типа `float`, в программе на Python для этого объекта выделяется память: в компьютерной архитектуре место расположения данного фрагмента памяти называется адресом памяти (*address*). Адрес объекта можно узнать при помощи стандартной функции `id()`:

```
print(id(24.5)) # 1974400231344
```

Это число является ссылкой на конкретное место в памяти, соответствующее фрагменту памяти, выделенному для хранения объекта типа `float` со значением `24.5`.

- Имя переменной может быть присвоено любому объекту («связано» с любым объектом) и использоваться для идентификации этого объекта в дальнейших вычислениях. Например:

```
a = 3
b = -0.5
print(a * b) # -1.5
```

# Имена переменных

Несколько правил, определяющих формирование допустимых («правильных») имен переменных:

- в именах переменных учитывается регистр символов (букв): `а` и `А` – это разные имена переменных;
- имена переменных могут содержать любую букву, символ подчеркивания `_` и любую цифру (`0 – 9`), но **не должны** начинаться с цифры;
- имя переменной не должно совпадать с одним из зарезервированных ключевых слов:

and	as	assert	async	await	break
class	continue	def	del	elif	else
except	finally	for	from	global	if
import	in	is	lambda	nonlocal	not
or	pass	raise	return	try	while
with	yield	False	True	None	

# Имена переменных

Некоторые соглашения по стилю:

- имена переменных должны быть осмысленными (`area` лучше, чем `a`), но не слишком длинными (`the_area_of_the_triangle` – это слишком громоздкое имя);
- в общем случае лучше не использовать `I` (буква `i` в верхнем регистре), `l` (буква `L` в нижнем регистре) и букву `o` в верхнем регистре, так как они очень похожи на цифры `1` и `0`;
- имена переменных `i`, `j` и `k`, как правило, используются для целочисленных счетчиков;
- рекомендуется использовать имена с буквами нижнего регистра с разделением слов символами подчеркивания вместо стиля именования «*CamelCase*»: например, `mean_height`, а не `MeanHeight`.

Эти и многие другие правила и соглашения определены в руководстве по стилю под названием `PEP8`, которое представляет собой одну из частей документации Python.

# Операции сравнения и логические операции



# Логический тип данных

- Для логического типа данных `bool` можно объявлять логические переменные, инициализируя их логическими значениями или присваивая им результат вычисления логических выражений.
- Логических констант в Python две: `True` (истина) и `False` (ложь).

```
x = True  
y = False  
z = 2 > -1  
print(x, y, z) # True False True
```

# Операции сравнения

Логические выражения являются аналогом математической формулы, результатом его вычисления будет одна из двух логических констант – `True` или `False`.

Операция	Описание
<code>x &lt; y</code>	Меньше
<code>x &lt;= y</code>	Меньше или равно
<code>x &gt; y</code>	Больше
<code>x &gt;= y</code>	Больше или равно
<code>x == y</code>	Равно
<code>x != y</code>	Не равно

Приоритет операций сравнения **ниже**, чем у арифметических операций:

```
print(2 + 3 * 5 > 7 / 2 + 3.5) # True
```

Операции сравнения сравнивают относительные величины operandов и возвращают результат логического типа:

```
# Больше или равно: int приводится к float
print(2.0 >= 1) # True
# Значения равны
print(3.0 == 3.0) # True
# Значения не равны
print(3.0 != 3.0) # False
```

# Операции сравнения

Особое внимание требуется при сравнении чисел с плавающей точкой на равенство. Поскольку числа с плавающей точкой хранятся с некоторой неточностью, вычисления с ними часто приводят к потере точности, и можно получить неожиданные результаты по неосторожности:

```
a = 0.01
b = 0.1 ** 2
print(a == b) # False
```

Результат возведения в квадрат представления числа `0.1` в виде числа с плавающей точкой равен `0.0100000000000002`, поэтому результат операции сравнения равен `False`.

В библиотеке `math` есть функция `isclose()`, которая проверяет равенство двух чисел с плавающей точкой с учетом некоторого абсолютного или относительного интервала допустимого отклонения:

```
x = math.isclose(0.1 ** 2, 0.01)
print(x) # True
```

## Логическая операция `or`

- Бинарная логическая операция `or` называется логическим сложением (логическое «ИЛИ»).
- Результатом операции `or` будет `False` только тогда, когда оба логических выражения будут иметь значения `False`, а в прочих случаях результатом будет `True`.

```
print(9 + 3 < 10 or 2 + 2 == 4) # True  
print(4 + 2 < 0 or 10 * 2 >= 100) # False
```

Если выражение, стоящее слева от оператора `or` вернет `True`, то второе выражение вычисляться не будет:

```
x = 10  
y = 0  
print(x or y) # 10
```

поэтому сложные проверки условий лучше ставить после оператора `or`.

## Логическая операция `and`

- Бинарная логическая операция `and` называется логическим умножением (логическое «И»).
- Результатом операции `and` будет `True` только тогда, когда оба логических выражения будут иметь значения `True`, а в прочих случаях результатом будет `False`.

```
print(9 + 3 < 10 and 2 + 2 == 4) # False  
print(4 + 2 != 0 and 10 * 2 == 20) # True
```

Аналогично оператору `or`, Python не будет вычислять выражение справа от оператора `and`, если первое выражение вернуло `False`:

```
x = 10  
y = 0  
print(y > x and x > y) # False
```

# Логическая операция `not`

- Унарная логическая операция `not` называется логическим отрицанием («НЕ», инверсия) и указывается перед логическим выражением для получения его противоположного значения.
- Приоритет операции `not` ниже, чем у операций сравнения, поэтому следующее за ней логическое выражение можно не заключать в скобки:

```
print(3 + 5 >= 8) # True  
print(not 3 + 5 >= 8) # False
```

- В составных выражениях первыми выполняются операторы сравнения, а затем логические операторы:

```
print(not 7.5 < 0.9 or 4 == 4) # True
```

- Приоритет можно изменить при помощи круглых скобок:

```
print(not (7.5 < 0.9 or 4 == 4)) # False
```

# Приведение к логическому типу

- Конструктор типа `bool(x)` может использоваться для явного приведения любого значения к логическому типу (если это значение можно интерпретировать как логический тип).
- Если аргумент `x` ложь или опущен, то будет возвращено значение `False`.

```
print(bool(1), bool(-1.0)) # True True
print(bool(0), bool(0.0)) # False False
print(bool()) # False
```

- В выражениях с логическими операторами может происходить неявное приведение объектов к логическому типу:

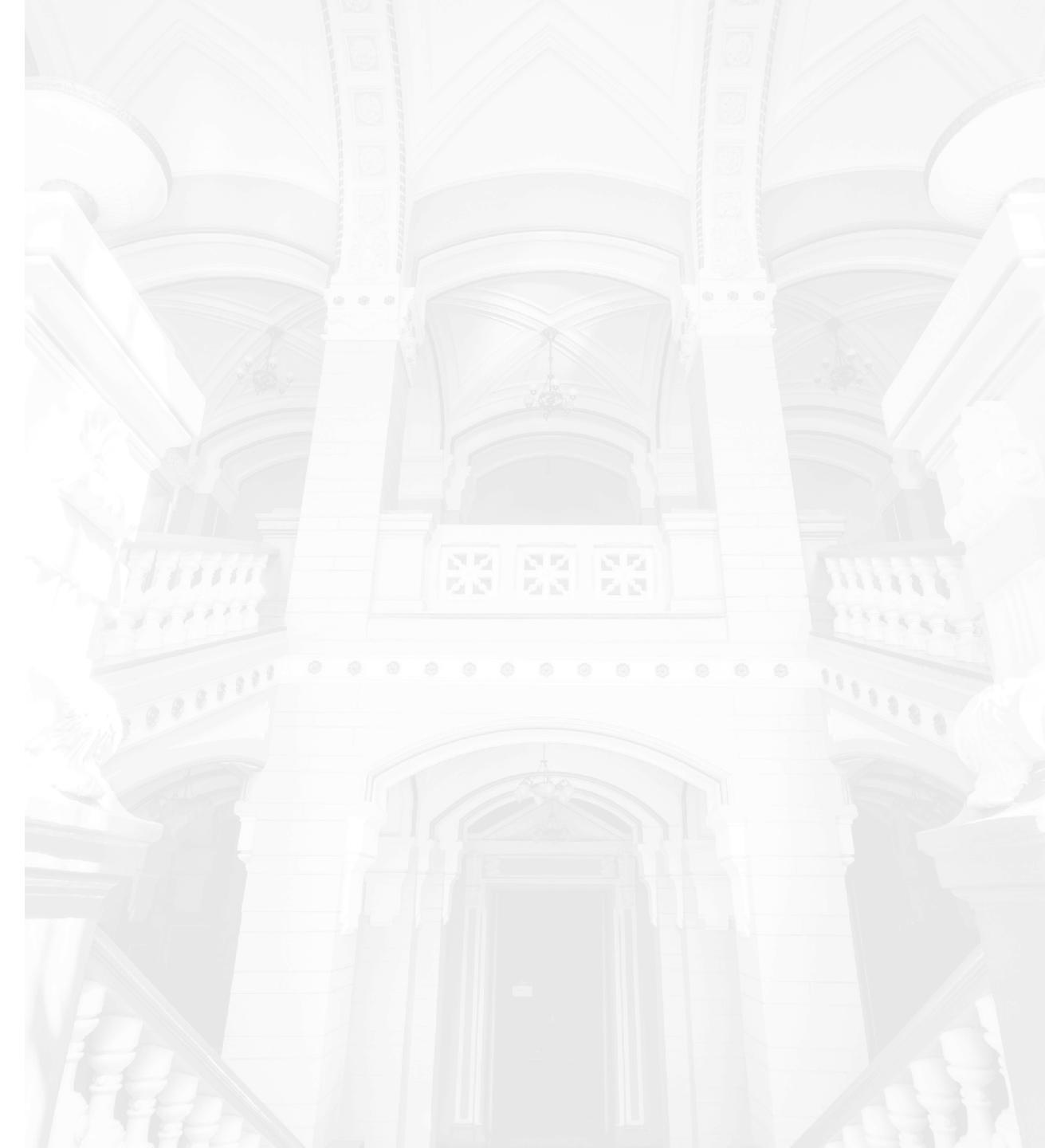
```
a = 0
print(a - 2 or a) # -2
print(4 > 3 and a - 2) # -2
print(4 > 3 and a) # 0
print(a and 4 > 3) # 0
```

# Сцепленные операции сравнения

- В Python также есть возможность выстраивать цепочки из нескольких операторов сравнения для выполнения проверок вхождения в диапазон.
- Сцепленные сравнения являются краткой записью для более массивных булевых выражений.

```
print(x < y > z)  # False
print(x < y and y > z)  # False
print(3 < 6 < 9.0 < 12)  # True
print(3 > 6 > 9 > 12)  # False
```

# Строки



# Строки (str)

- Строки в Python относятся к **неизменяемым последовательностям**, что говорит о том, что содержащиеся в них символы имеют позиционный порядок слева направо и не могут быть изменены на месте.
- Строки в Python исполняют ту же роль, что и массивы символов в С-подобных языках, но в сравнении с массивами, они обладают инструментарием более высокого уровня.
- Строки в Python снабжены мощным набором инструментов обработки. Кроме того, в отличие от языков, подобных C, в Python не предусмотрен отдельный тип для индивидуальных символов, вместо этого применяются односимвольные строки.
- В плане обработки строки поддерживают операции **выражений**, такие как конкатенация (объединение строк), срезы (извлечение частей), индексация и т.д. Кроме операций выражений Python предоставляет набор **методов** строк, которые реализуют общие задачи, специфичные для строк.

## Синтаксис строк

Python предлагает довольно обширный набор инструментов для работы со строками и в большинстве случаев использование строк проходит довольно легко. Вот несколько возможных способов записи строк в Python-коде:

- одинарные кавычки – `'spam'`
- двойные кавычки – `"spam"`
- тройные кавычки – `"""... spam ..."""`
- неформатированные строки – `r'C:\new\test.spm'`
- прочие способы записи строк.

Формы с одинарными и двойными кавычками, безусловно, являются наиболее популярными, другие используются в особых случаях.

# Операции над строками

1. Встроенная функция `len()` возвращает длину строки:

```
print(len('abc')) # 3
```

2. **Конкатенация** (сложение) строк выполняется при помощи операции `+` и создает новый объект строки с объединенным содержимым ее operandов:

```
print('abc' + 'def') # abcdef
```

3. Повторение выполняется при помощи операции `*` и идентично добавлению строки к самой себе несколько раз:

```
# Повторение: то же, что 'Hi!' + 'Hi!' + ...
print('hi' * 4) # Hi!Hi!Hi!Hi!
nC10 = 'CH3-' + 'CH2-' * 8 + 'CH3'
print(nC10) # CH3-CH2-CH2-CH2-CH2-CH2-CH2-CH2-CH2-CH3
```

# Сравнение строк

- Сравнение строк происходит последовательно: первый символ одной строки сравнивается с первым символом другой. Если они равны, сравниваются символы на следующей позиции.
- Для сравнения строк используются операторы <, <=, >, >=, ==, !=

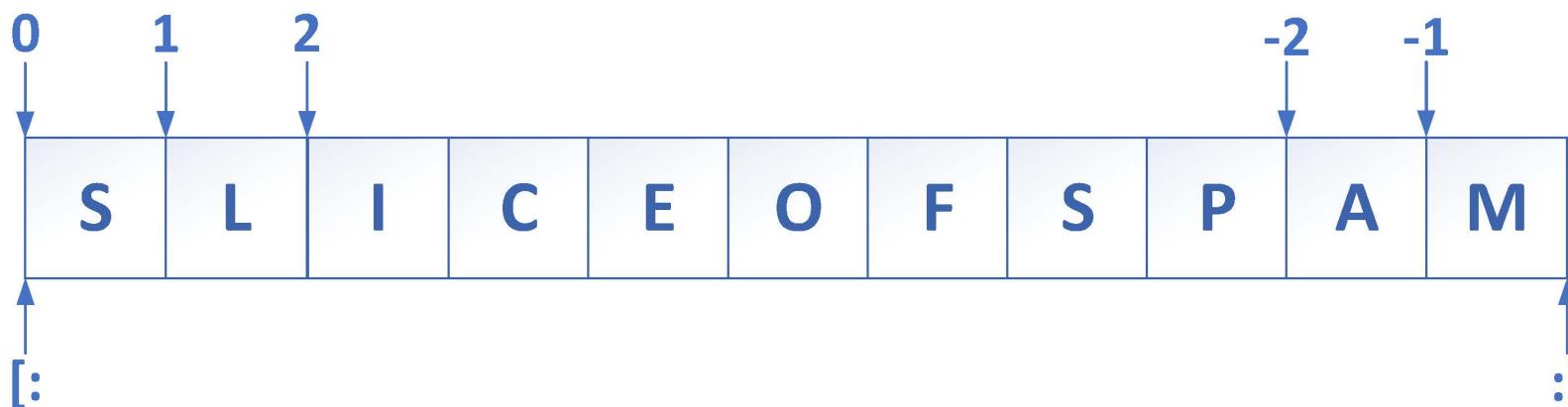
```
# 'p' > 'P'  
print('python' > 'Python') # True  
print(ord('p'), ord('P')) # 112 80
```

- Проверка на **идентичность** проводиться при помощи оператора `is`: если имена указывают на один объект – оператор `is` вернет `True`, в противном случае – `False`:

```
s1 = 'Python is the best!'  
s2 = s1 # Создаем синоним s1: ссылка на один и тот же объект строки  
s3 = 'Python is the best!' # Создаем новый объект  
print(s1 == s2 == s3) # True  
print(s1 is s2) # True  
print(s1 is s3) # False  
print(id(s1), id(s3)) # 1971522386560 1971509687280
```

# Операции индексации

- Строки являются упорядоченными коллекциями символов и поэтому поддерживают доступ к своим элементам по индексу.
- Индексация** – предоставление индекса желаемого компонента в квадратных скобках после имени, с которым связан объект строки. Результатом будет являться односимвольная строка в указанной позиции.
- Индексы в Python начинаются с `0` и заканчиваются величиной, на единицу меньше, чем длина строки.
- Python разрешает получать элементы из последовательностей с использованием **отрицательных** индексов.



# Операции срезов

- Срезы – обобщенная форма индексации для получения целого **сегмента** вместо одиночного элемента.
- При выполнении среза Python извлекает элементы, начиная с нижней границы и заканчивая, но не включая верхнюю границу, и возвращает новый объект, содержащий извлеченные элементы.
- Если левая и/или правая границы не указаны, по умолчанию для них принимаются индексы `0` и длина последовательности, соответственно.

```
s = 'chemistry'  
print(s[1:3], s[1:], s[:-1]) # he hemistry chemistr
```

## Расширенные срезы

- В Python для выражений срезов есть поддержка опционального третьего индекса, используемого в качестве **шага**;
- Шаг прибавляется к индексу каждого извлеченного элемента;
- Полная форма среза выглядит следующим образом:

$$x[i : j : k]$$

что означает «извлечь элементы из `x`, начиная с индекса `i` и заканчивая индексом `j-1`, с шагом `k`»;

- Третий предел, `k`, по умолчанию, равен `+1` и поэтому все элементы в срезе обычно извлекаются слева направо. Однако если указать явное значение, то можно применить третий предел для пропуска элементов или смены порядка их следования на противоположный.

## Расширенные срезы

- Например, `x[1:10:2]` вернет каждый второй элемент из `x` в рамках индексов 1-9, т.е. элементы с индексами 1, 3, 5, 7 и 9.
- По аналогии, верхний и нижний пределы по умолчанию принимаются равными `0` и длине последовательности, соответственно, поэтому `x[::-2]` вернет каждый второй элемент с начала и до конца последовательности:

```
s = 'Beautifulisbetterthanugly'  
# Пропуск элементов  
print(s[1:10:2]) # euiui  
print(s[::-2]) # Batflsetrhngy
```

# Расширенные срезы

- Можно также использовать отрицательный шаг для получения элементов в обратном порядке. Например, выражение среза `'spam'[::-1]` вернет новую строку `'maps'` – шаг `-1` указывает, что срез должен идти справа налево, а не слева направо:

```
s = 'spam'  
# Смена порядка элементов на противоположный  
print(s[::-1]) # maps
```

- При отрицательном шаге смысл нижней и верхней границ по сути меняется на противоположный. Таким образом, срез `x[5:1:-1]` получает элементы со второго по пятый в обратном порядке (элементы с индексами 5, 4, 3 и 2):

```
s = 'Simpleisbetterthancomplex'  
# Смысл границ изменяется  
print(s[5:1:-1]) # elpm
```

# Основные методы строк

Метод	Описание
center(width)	Возвращает строку, отцентрированную в новую строку с общим количеством символов width
endswith(suffix)	Возвращает True , если строка заканчивается подстрокой suffix
startswith(prefix)	Возвращает True , если строка начинается подстрокой prefix
index(substring)	Возвращает наименьший индекс в строке, соответствующий содержащейся в ней подстроке substring
upper()	Возвращает копию строки, в которой все символы переведены в верхний регистр
lower()	Возвращает копию строки, в которой все символы переведены в нижний регистр

# Основные методы строк

Метод	Описание
<code>title()</code>	Возвращает копию строки, в которой все слова начинаются с заглавных букв, а все прочие символы переведены в нижний регистр
<code>replace(old, new)</code>	Возвращает копию строки, в которой каждая подстрока <code>old</code> заменена подстрокой <code>new</code>
<code>split([sep])</code>	Возвращает список подстрок из исходной строки, которые разделены заданной строкой <code>sep</code> . Если строка <code>sep</code> не задана, то разделителем является любое количество пробелов
<code>join([list])</code>	Использует строку как разделитель при объединении списка <code>list</code> строк
<code>isalpha()</code>	Возвращает <code>True</code> , если все символы в строке являются алфавитными и строка не пустая, иначе возвращается <code>False</code>

# Примеры использования методов строк

```
a = 'java python c++ fortran '
print(a.isalpha) # False
b = a.title()
print(b) # Java Python C++ Fortran
```

```
c = b.replace(' ', '!\\n')
print(c)
# Java!
# Python!
# C++!
# Fortran!
```

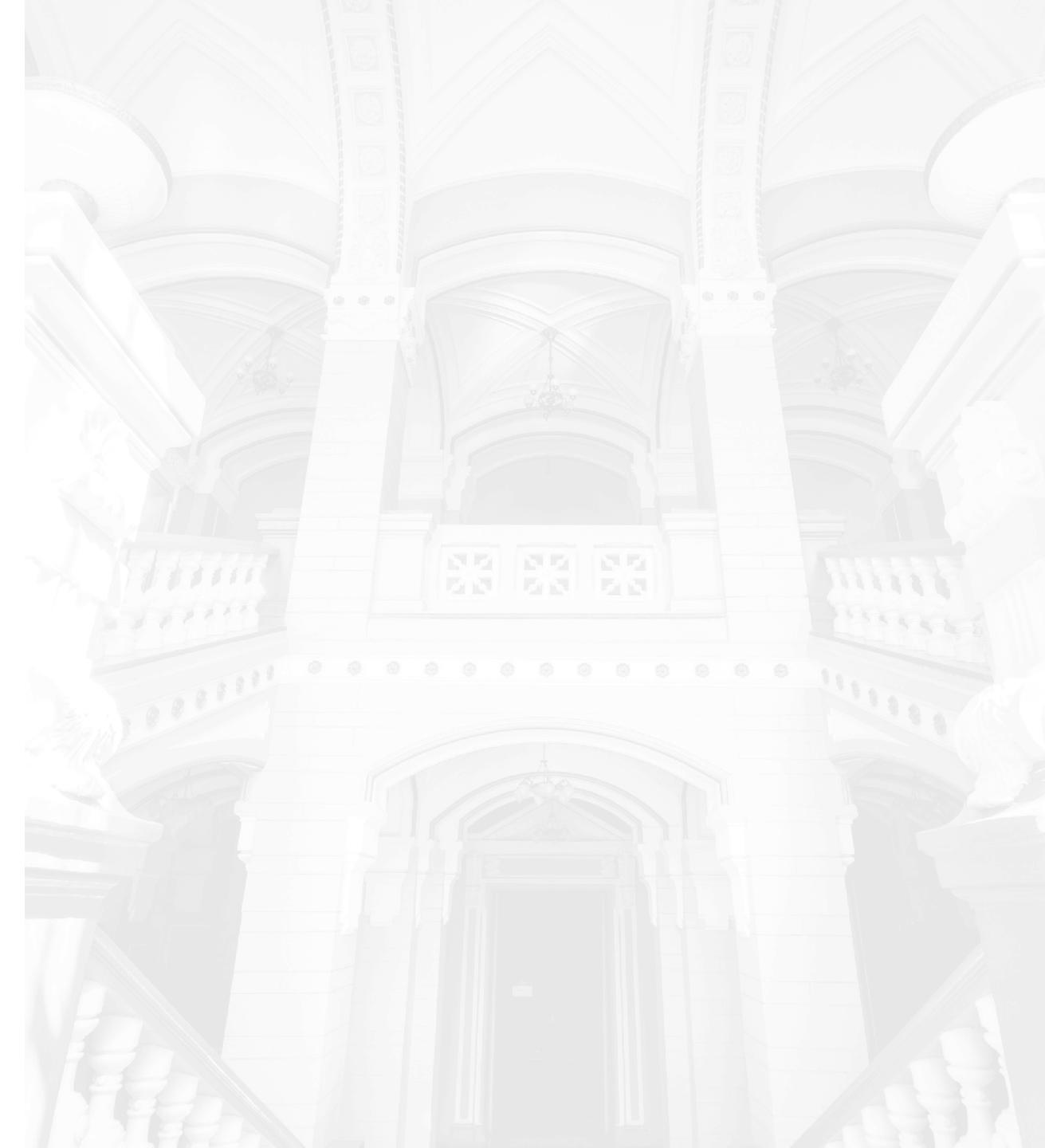
```
i = c.index('Python')
print(i) # 6
print(c[6:]).startswith('Py')) # True
print(c[6:12].isalpha()) # True
```

# Форматированные строки (f-строки)

- Литерал форматированных строк или f-строки – это строковый литерал с префиксом `f` или `F`. Данные строки могут содержать замещающие поля, которые являются выражениями в фигурных скобках `{}`.
- Мини-язык для спецификатора формата такой же, как и в методе `format()`.

```
print(f'{1.2354:.2f}') # 1.24
print(f'{1.2354:e}') # 1.235400e+00
print(f'{1.2354:.3e}') # 1.235e+00
print(f'{1.2354:g}') # 1.2354
```

# Списки



# Списки (list)

- Списки в Python – наиболее гибкая разновидность объектов упорядоченных коллекций.
- Списки могут содержать объекты любого типа: строки, числа или другие списки.
- Списки **можно изменять** на месте присваиванием по индексам, с использованием срезов, вызвав специальные методы или выполнив оператор удаления.

Операция	Описание
<code>a = []</code> <code>a = list()</code>	Создание пустого списка
<code>a = [123, 'abc', 1.354, []]</code>	Четыре элемента: индексы 0...3
<code>a = ['Joe', 30.0, ['dev', 'prof']]</code>	Вложенные списки
<code>a = list('hello')</code>	Список элементов итерируемого объекта
<code>a = list(range(-5, 6))</code>	Список последовательных целых чисел
<code>a[i]</code>	Индекс

# Списки (list)

Операция	Описание
<code>a[i][j]</code>	Индекс вложенного списка
<code>a[i:j]</code>	Срез
<code>len(a)</code>	Длина
<code>a1 + a2</code>	Конкатенация
<code>a * 3</code>	Повторение
<code>x in a</code>	Вхождение
<code>a.append(5)</code>	Добавление элемента в конец списка
<code>a.extend([10, 20, 30])</code>	Добавление списка в конец исходного списка

# Генераторы списков (list comprehension)

Генераторы списков – это способ создания нового списка с применением выражения к каждому элементу последовательности (по факту в любом итерируемом объекте).

```
chars = [c * 4 for c in 'HELLO']
print(chars) # ['HHHH', 'EEEE', 'LLLL', 'LLLL', '0000']
```

- Генераторы списков записываются более кратко и выполняются чуть быстрее.
- В сложных случаях лучше использовать цикл `for` из-за его более высокой читаемости.

```
chars = []

for c in 'HELLO':
    chars.append(c * 4)

print(chars) # ['HHHH', 'EEEE', 'LLLL', 'LLLL', '0000']
```

# Индексация и срезы

- Индексация и срезы для списков работают аналогично тому, как это было описано для объектов строк.
- Результатом индексации списка может быть объект любого типа, находящийся по указанному индексу, тогда как срезы всегда возвращают новый объект списка.

```
fruits = ['banana', 'orange', 'apple']

# Индексы начинаются с нуля
print(fruits[2]) # 'apple'

# Отрицательные индексы отсчитываются справа
print(fruits[-2]) # 'orange'

# Срезы получают сегменты
print(fruits[1:3]) # ['orange', 'apple']

# Результат среза всегда новый список
print(fruits[-1:]) # ['apple']
```

## Вложенность списков

- Внутри списков могут содержаться вложенные списки или объекты других типов.
- Матрицы в Python можно представить в виде вложенных списков. Пример матрицы  $3 \times 3$ :

```
matrix = [[10, 20, 30], [40, 50, 60], [70, 80, 90]]
```

- Если указать один индекс, то будет получена целая строка, а при указании двух индексов будет возвращен элемент строки:

```
print(matrix[1]) # [40, 50, 60]
print(matrix[2][0]) # 70
matrix = [[10, 20, 30],
          [40, 50, 60],
          [70, 80, 90]]
print(matrix[1][1]) # 50
```

# Изменение списков

- Так как списки – **изменяемый** тип объектов, для них определены операции, которые могут модифицировать объект списка на месте.
- Операции модификации списков изменяют объект списка напрямую, перезаписывая его старое значение, без необходимости создания новой копии, как в случае работы со строками.

## Присваивание по индексам и срезам

```
food = ['burger', 'pizza', 'buritto']

food[1] = 'toast' # Присваивание по индексу
print(food) # ['burger', 'toast', 'buritto']

food[:2] = ['need', 'more'] # Присваивание срезу
print(food) # ['need', 'more', 'buritto']
```

# Примеры присваиваний по срезу

```
a = [10, 20, 30]

a[1:2] = [4, 5] # Замена/вставка
print(a) # [10, 4, 5, 30]

a[1:1] = [60, 70] # Вставка (ничего не заменяется)
print(a) # [10, 60, 70, 4, 5, 30]

a[1:2] = [] # Удаление (ничего не добавляется)
print(a) # [10, 70, 4, 5, 30]

a[:0] = [1, 2, 3] # Вставка на место :0, пустой срез в начале
print(a) # [1, 2, 3, 10, 70, 4, 5, 30]

a[len(a):] = [4, 5, 6] # Вставка на место len(a):, пустой срез в конце
print(a) # [1, 2, 3, 10, 70, 4, 5, 30, 4, 5, 6]
```

## Вызовы методов списков

- Подобно строкам, списки имеют набор специфичных методов, многие из которых ведут к изменению исходного списка на месте:

```
a = ['eat', 'more', 'SPAM']
a.append('please')
print(a) # ['eat', 'more', 'SPAM', 'please']
```

- Наиболее распространенный метод – `append()` добавляет объект в конец списка.
- Эффект выполнения выражения `a.append(x)` аналогичен `a + [x]` с одним принципиальным отличием: первый вариант изменяет `a` на месте, а второй вариант создает новый объект списка.

# Вызовы методов списков

- Метод `extend()` добавляет множество элементов в конец списка;
- Метод `pop()` удаляет и возвращает последний элемент списка, если не указан индекс;
- Метод `reverse()` изменяет порядок элементов в списке на противоположный (обращает список) на месте.

```
a = [0, 1]

a.extend([2, 3, 4]) # Добавление множества элементов в конец списка
print(a) # [0, 1, 2, 3, 4]

x = a.pop() # Удаление и возврат последнего элемента
print(x) # 4
print(a) # [0, 1, 2, 3]

a.reverse() # Метод обращения списка на месте
print(a) # [3, 2, 1, 0]
```

# Вызовы методов списков

- `index()` – нахождение индекса элемента;
- `insert()` – вставка элементов по индексу;
- `remove()` – удаление элементов списка;
- `pop(i)` – удаляет и возвращает элемент по индексу `i`;
- `count()` – подсчет количества вхождений.

```
a = ['spam', 'eggs', 'ham']

i = a.index('spam') # Поиск индекса объекта
print(i) # 0

a.insert(1, 'toast') # Вставка по индексу
print(a) # ['spam', 'toast', 'eggs', 'ham']

a.remove('spam') # Удаление по значению
print(a) # ['toast', 'eggs', 'ham']
```

```
x = a.pop(1) # Удаление по индексу
print(x) # 'eggs'
print(a) # ['toast', 'ham']

c = a.count('ham') # Количество вхождений
print(c) # 1
```

## Метод `sort()`

- Для сортировки элементов списка по возрастанию достаточно вызвать метод `sort()`.
- Метод `sort()` изменяет исходный список (и возвращает `None`):

```
a = [5, 2, 3, 1, 4]
a.sort()
print(a) # [1, 2, 3, 4, 5]

a = [10, 5, 4, 20, 50]
x = a.sort()
print(x) # None
print(a) # [4, 5, 10, 20, 50]
```

## Метод sort()

- Отсортировать можно те списки, элементы которых можно сравнить друг с другом:

```
a = ['a', 'B', 'C', 'b', 'c', 'A']
a.sort()
print(a) # ['A', 'B', 'C', 'a', 'b', 'c']

a = [[1, 2, 3], [3, 2, 1], [2, 3, 1]]
a.sort()
print(a) # [[1, 2, 3], [2, 3, 1], [3, 2, 1]]

a = [1, [20.5, 10], 'A', -5]
a.sort() # TypeError: '<' not supported between instances of 'list' and 'int'
```

## Дополнительные сведения о методе `sort()`

- Аргумент `reverse` позволяет сортировку по убыванию, а параметр `key` принимает функцию с одним аргументом, возвращающую значение, используемое при сортировке.

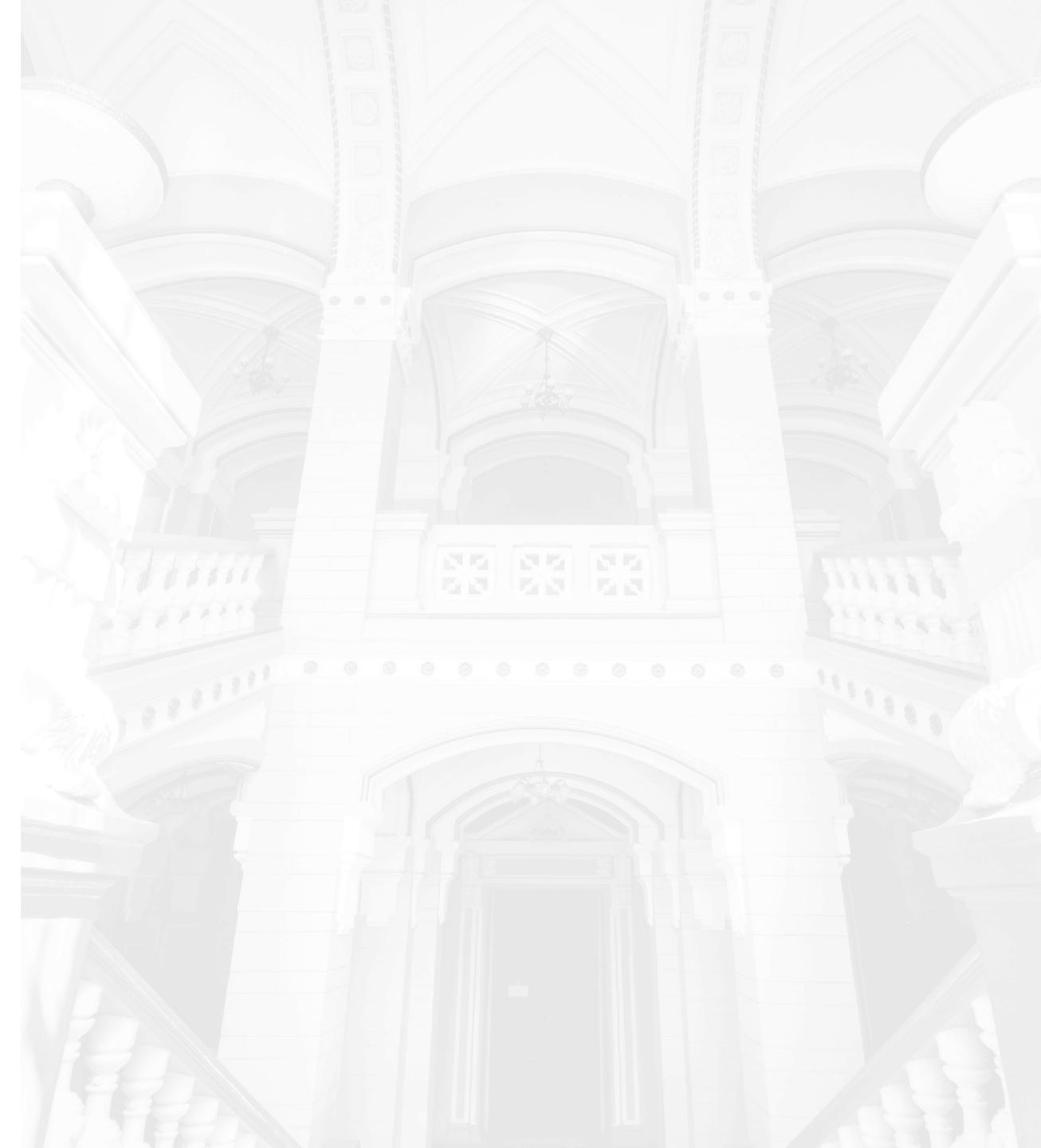
```
a = [-10, 5, 20, -3, 10]
a.sort() # Сортировка с учетом знака
print(a) # [-10, -3, 5, 10, 20]

a.sort(reverse=True) # Сортировка в порядке убывания
print(a) # [20, 10, 5, -3, -10]

a.sort(key=abs) # Сортировка без учета знака
print(a) # [-3, 5, -10, 10, 20]

a.sort(key=abs, reverse=True) # Изменение порядка сортировки
print(a) # [20, -10, 10, 5, -3]
```

# Кортежи



# Кортежи (tuple)

- Кортежи служат для хранения нескольких объектов вместе.
- Аналог списков, но без обширной функциональности. Кортежи **неизменяемы**.

Операция	Интерпретация
<code>t = ()</code> <code>t = tuple()</code>	Создание пустого кортежа
<code>t = (0, )</code>	Одноэлементный кортеж (не выражение)
<code>t = (0, 'Hi', 1.2, 3)</code>	Кортеж из четырех элементов
<code>t = 0, 'Hi', 1.2, 3</code>	Такой же кортеж, как в предыдущей строке
<code>t = ('John', ('prof', 'dev'))</code>	Вложенные кортежи
<code>t = tuple('hello')</code>	Кортеж из элементов итерируемого объекта
<code>t[i]</code>	Индекс

# Кортежи (tuple)

Операция	Интерпретация
<code>t[i][j]</code>	Индекс вложенного кортежа
<code>t[i:j]</code>	Срез
<code>len(t)</code>	Длина кортежа
<code>t1 + t2</code>	Конкатенация
<code>t * 3</code>	Повторение
<code>'spam' in t</code>	Проверка вхождения
<code>t.index('Hi')</code>	Поиск индекса элемента
<code>t.count('hello')</code>	Подсчет повторений элемента

# Операции над кортежами

- Кортежи поддерживают обычные операции, специфичные для последовательностей:

```
# Конкатенация  
print((10, 20) + (30, 40)) # (10, 20, 30, 40)
```

```
# Повторение  
print((1, 2) * 5) # (1, 2, 1, 2, 1, 2, 1, 2)
```

```
# Сравнение  
print((10, 20) < (20, 30) < (30, 40)) # True
```

```
t = (10, 20, 30, 40, 50)
```

```
# Индексация, срезы  
print(t[0], t[1:3]) # 10 (20, 30)
```

# Операции над кортежами

- Создание кортежа из одного элемента:

```
x = (30) # Целое число
print(x) # 30
print(type(x)) # <class 'int'>
y = (30,) # Кортеж, содержащий целое число
print(y) # (30,)
print(type(y)) # <class 'tuple'>
```

- Сортировка кортежа

```
t = ('b', 'c', 'a', 'e', 'd')
tmp = list(t) # Создание списка из элементов кортежа
tmp.sort() # Сортировка списка
print(tmp) # ['a', 'b', 'c', 'd', 'e']
t = tuple(tmp) # Создание кортежа из элементов списка
print(t) # ('a', 'b', 'c', 'd', 'e')
```

# Преобразование кортежей

- Преобразовать элементы кортежа можно при помощи выражения генератора списка, либо при помощи итерации по его элементам в цикле `for` :

С использованием генератора списка:

```
t = (1, 2, 3, 4, 5)

a = [x ** 3 for x in t]
print(a) # [1, 8, 27, 64, 125]

t = tuple(a)
print(t) # (1, 8, 27, 64, 125)
```

С использованием цикла `for`:

```
t = (1, 2, 3, 4, 5)

a = []
for number in t:
    a.append(number ** 3)
print(a) # [1, 8, 27, 64, 125]

t = tuple(a)
print(t) # (1, 8, 27, 64, 125)
```

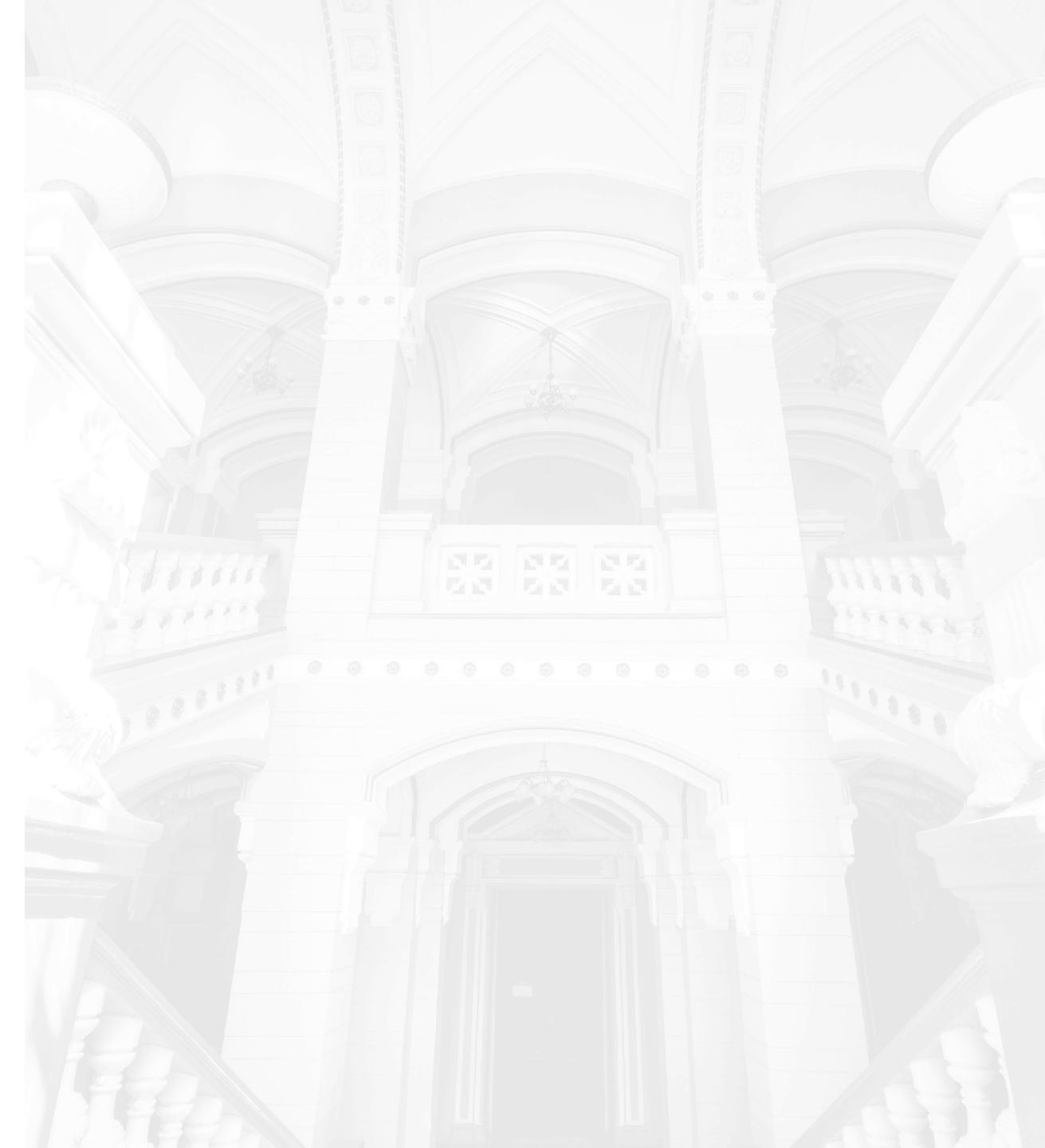
# Преобразование кортежей

Изменять элементы кортежей по индексу, как и элементы строк, **нельзя**:

```
t = (1, 2, 3, 4, 5)
t[1] = 'hi!' # TypeError: 'tuple' object does not support item assignment

s = 'compound'
s[3] = 'b' # TypeError: 'str' object does not support item assignment
```

# Операторы управления потоком команд



# Конструкция `if...elif...else`

Конструкция `if ... elif ... else` позволяет выполнять инструкции в зависимости от выполнения заданных условий, т. е. в зависимости от результата одной или нескольких проверок логических выражений (которые при вычислении выдают логические значения `True` или `False`):

```
if <логическое выражение 1>:  
    <блок конструкций 1>  
  
elif <логическое выражение 2>:  
    <блок инструкций 2>  
  
elif <логическое выражение 3>:  
    <блок инструкций 3>  
  
...  
  
else:  
    <блок инструкций>
```

- Если вычисление `<логическое выражение 1>` дает результат `True`, то выполняется `<блок инструкций 1>`, иначе если при вычислении `<логическое выражение 2>` получен результат `True`, то выполняется `<блок инструкций 2>` и т.д.
- Если ни одно из заданных явно логических выражений не дало результат `True`, то выполняется блок инструкций, следующих за ключевым словом `else`.

Блоки `elif` и `else` являются необязательными.

## Конструкция `if...elif...else`

Все эти блоки инструкций сдвигаются вправо с помощью 4-х пробелов:

```
for x in range(10):
    if x <= 3:
        print(x, 'is less than or equal to three')
    elif x > 5:
        print(x, 'is greater than five')
    else:
        print(x, 'must be four or five')
```

```
0 is less than or equal to three
1 is less than or equal to three
2 is less than or equal to three
3 is less than or equal to three
4 must be four or five
5 must be four or five
6 is greater than five
7 is greater than five
8 is greater than five
9 is greater than five
```

## Конструкция `if...elif...else`

В действительности выражение проверки условия при вычислении не обязательно должно выдавать только логические значения `True` или `False`: другие типы данных считаются равнозначными `True`, если они не равны `0` (`int`) или `0.` (`float`), пустой строке `''`, пустому списку `[]`, пустому кортежу `(,)` и т. д. или специальному значению `None`.

```
for x in range(10):
    if x % 2:
        print(x, 'is odd')
    else:
        print(x, 'is even')
```

Этот код работает, потому что `x % 2 = 1` для нечетных целых чисел, что равнозначно `True`, и `x % 2 = 0` для четных целых чисел, что равнозначно `False`.

# Тернарное выражение `if / else`

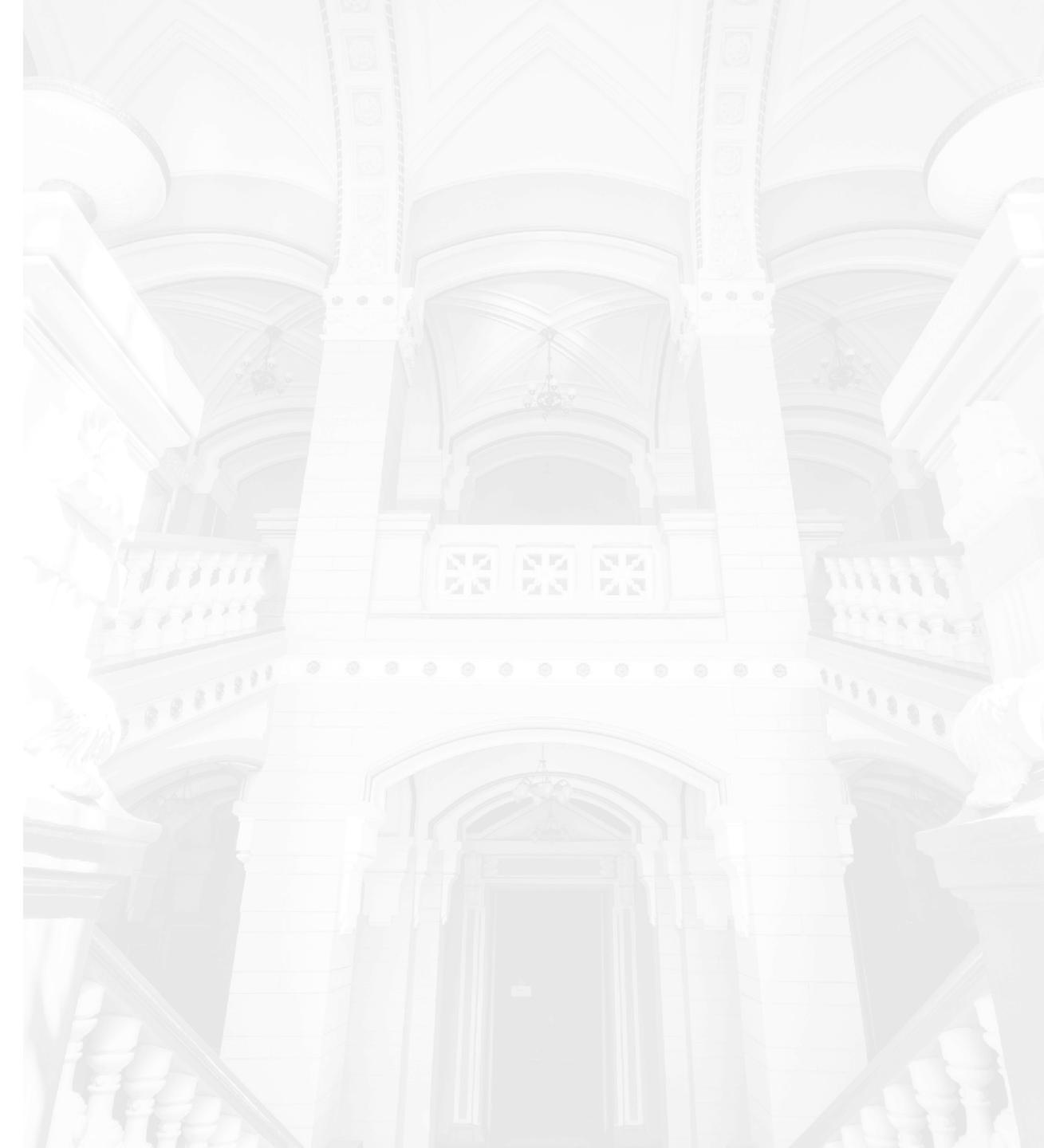
- Зачастую элементы, использованные в операторе `if`, достаточно просты, так что распространение такого оператора на четыре строки выглядит излишеством.
- В других случаях конструкцию подобного рода может понадобиться вложить в более крупный оператор, а не присваивать ее результат какой-то переменной.
- По указанным причинам в Python был добавлен новый формат условного выражения, который позволяет определить то же самое в одном действии (тернарный оператор).

```
if x:  
    a = y  
else:  
    a = z
```

```
a = y if x else z
```

Использовать тернарный оператор нужно крайне умеренно и только в тех случаях, когда все его составные части относительно просты, иначе предпочтительнее использовать форму полного оператора `if` для облегчения его будущего модифицирования.

# Операторы цикла в Python



# Операторы цикла

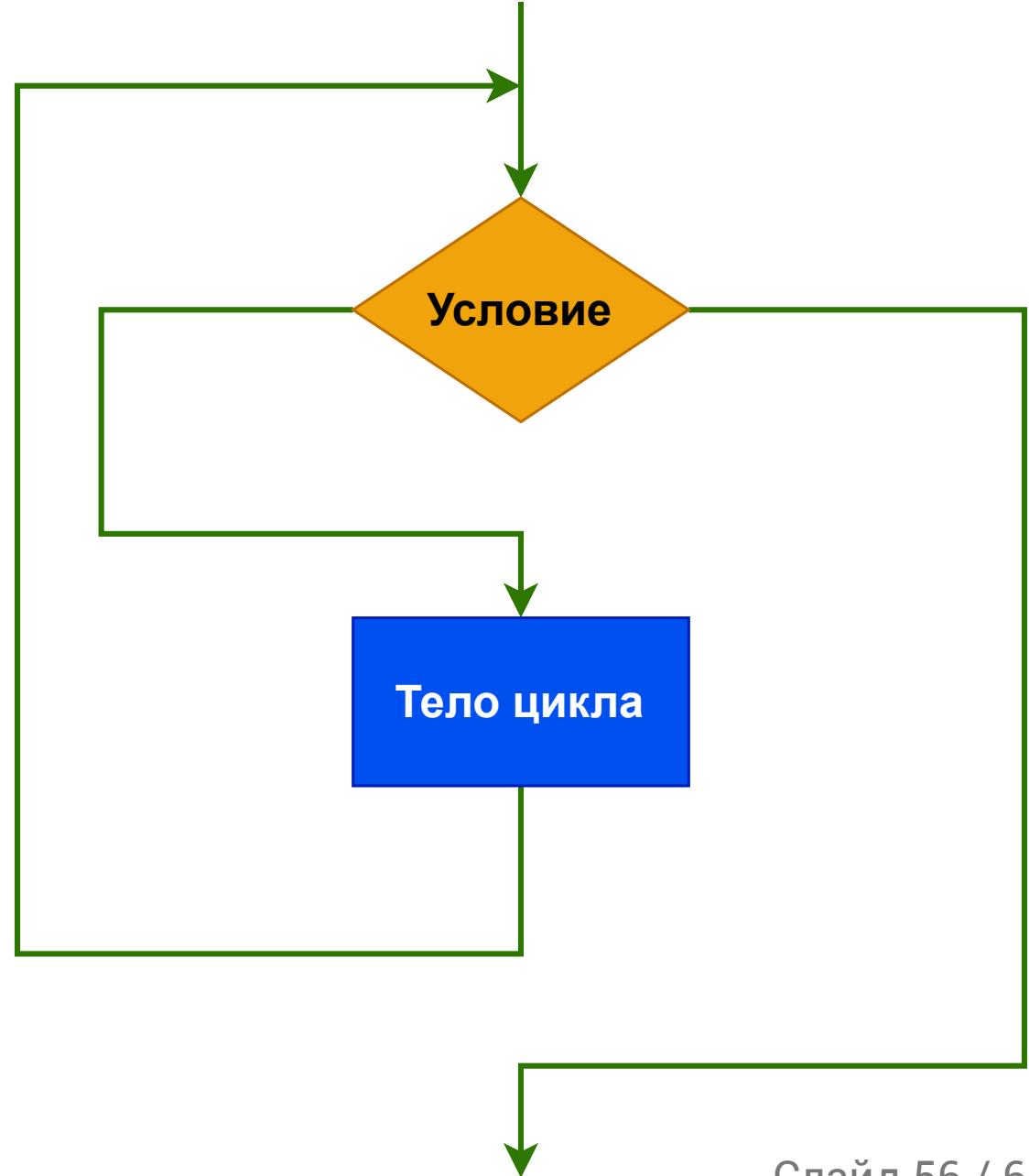
- Алгоритмы решения многих задач требуют некоторого количества повторений своих отдельных частей.
- Такие повторяющиеся участки называют циклическими, а операторы языка Python, реализующие соответствующие повторения – **операторами цикла**.
- Цикл состоит из **заголовка цикла** и **тела цикла**.
- Заголовок определяет условие прекращения (или выполнения) цикла, а тело цикла содержит операторы, которые нужно повторять.

Операторы цикла в Python:

1. Цикл `while`
2. Цикл `for`

# Оператор цикла `while`

- Оператор `while` многократно повторяет блок операторов до тех пор, пока проверка в заголовочной части оценивается как истина.
- Управление продолжает возвращаться к началу оператора, пока проверка не даст ложное значение. Когда результат проверки становится ложным, управление переходит на оператор, следующий после блока `while`.
- Если проверка оценивается в ложное значение с самого начала, тогда тело цикла никогда не выполнится и оператор `while` пропускается.



# Оператор цикла `while`

Общий формат цикла `while`:

```
while expression: # Проверка цикла
    operator_1 # Тело цикла
    operator_2
    ...
    operator_n
```

Цикл `while` можно использовать:

- в математических итерационных алгоритмах для проведения вычислений с заданной точностью;
- при вводе данных, когда их количество заранее неизвестно, а условие завершения ввода определено некоторым введенным значением;
- при поиске нужного элемента в какой-либо структуре данных.

## Примеры использования цикла `while`

Вывод чисел в диапазоне [0; 10) с шагом 1:

```
a, b = 0, 10  
  
while a < b:  
    print(a, end=' ')  
    a += 1  
  
# 0123456789
```

# Примеры использования цикла while

```
number = 24
running = True

while running:
    guess = int(input('Введите целое число: '))

    if guess == number:
        print('Поздравляем, Вы угадали!')
        running = False # это останавливает цикл while

    elif guess < number:
        print('Нет, загаданное число немного больше этого')

    else:
        print('Нет, загаданное число немного меньше этого')

print('Завершение.') # другие операторы программы

# Введите целое число: 12
# Введите целое число: 24
# Поздравляем, Вы угадали!
# Завершение.
```

# Оператор цикла `for`

- Оператор `for ... in` также является оператором цикла, который осуществляет итерацию по **последовательности** объектов, т.е. проходит через каждый элемент в последовательности.
- **Последовательность** – это упорядоченный или неупорядоченный набор элементов.
- Во многих случаях в заголовке цикла `for` используется функция `range()`, которая является генератором арифметических прогрессий:

```
for i in range(10): # 10 не включительно
    print(i, end=' ') # 0 1 2 3 4 5 6 7 8 9

for i in range(1, 11): # можно задать начальное значение
    print(i, end=' ') # 1 2 3 4 5 6 7 8 9 10

for i in range(0, 11, 2): # также можно менять шаг
    print(i, end=' ') # 0 2 4 6 8 10
```

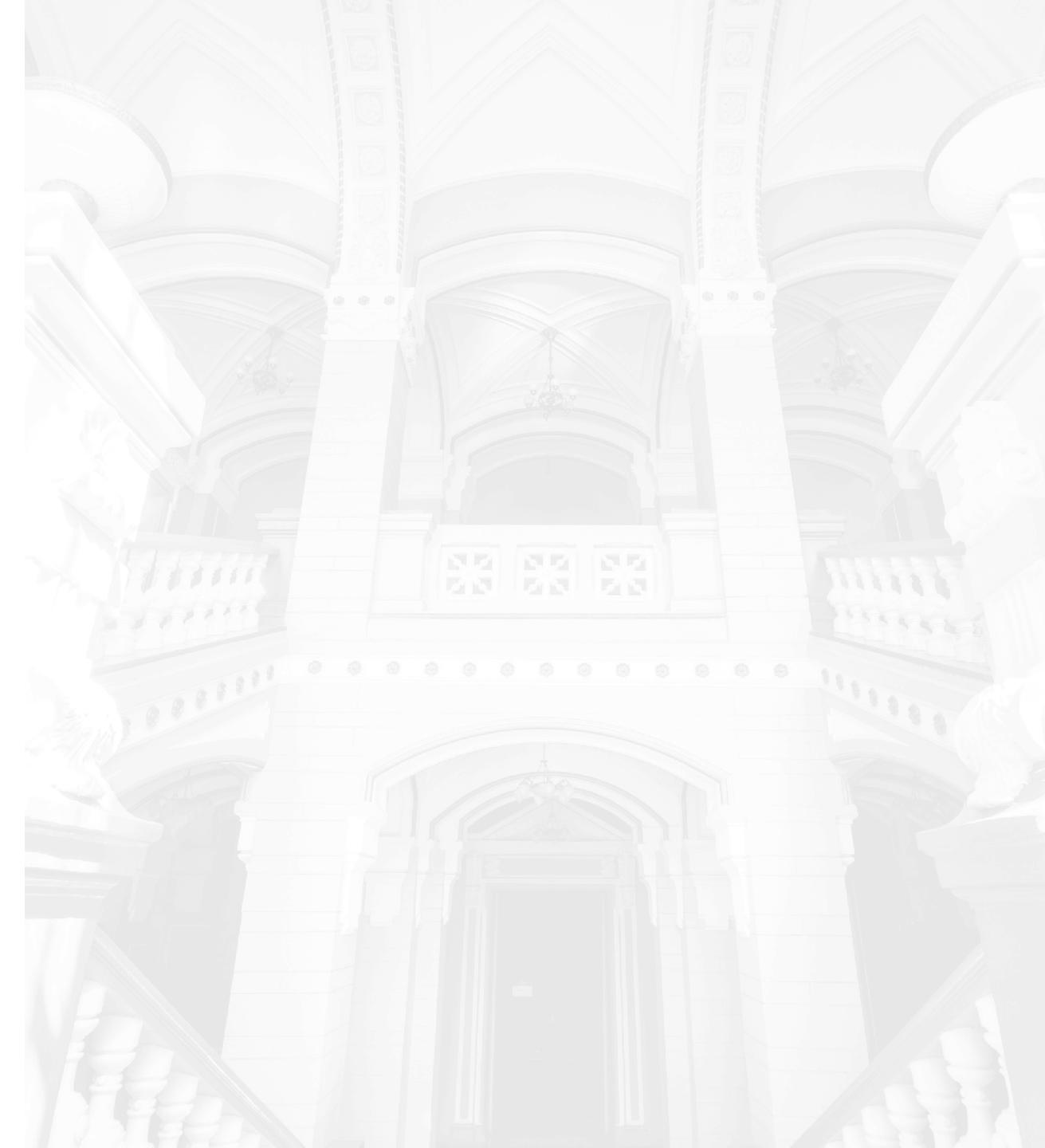
## Вложенные циклы `for`

Операторы цикла `for` могут быть вложены друг в друга на произвольную глубину:

```
for i in range(3):
    for j in range(3):
        if i != j:
            print(i, j, round(1 / (i + j), 2))

# 0 1 1.0
# 0 2 0.5
# 1 0 1.0
# 1 2 0.33
# 2 0 0.5
# 2 1 0.33
```

**Операторы  
`break`, `continue`,  
`pass` и  
конструкция `else`  
цикла**



# Оператор `break`

Оператор `break` выполняет немедленный выход из цикла, т.е. остановку выполнения команд, даже если условие выполнения цикла еще не приняло значение `False` или последовательность элементов не закончилась.

```
while True:  
    name = input('Enter name: ')  
    if name == 'stop':  
        break  
    age = input('Enter age: ')  
    print('Hello', name, '=>', int(age) * 2)  
  
# Enter name: John  
# Enter age: 35  
# Hello John => 70  
# Enter name: Julya  
# Enter age: 24  
# Hello Julya => 48  
# Enter name: stop
```

# Оператор `continue`

Оператор `continue` используется для немедленного перехода в начало цикла.

```
x = 10

while x:
    x -= 1

    if x % 2: # Нечетное? Тогда пропустить
        continue

    print(x, end=' ')

# 8 6 4 2 0
```

```
x = 10

while x:
    x -= 1

    if not x % 2: # Четное? Тогда выводим
        print(x, end=' ')

# 8 6 4 2 0
```

## Оператор `pass`

- Оператор `pass` – это заполнитель, обозначающий отсутствие действий, используемый в ситуациях, когда синтаксис требует оператора, но нет возможности выполнить что-либо полезное.
- Данный оператор часто применяется для кодирования пустого тела для составного оператора.

К примеру, с помощью `pass` можно написать бесконечный цикл, который на каждом проходе ничего не делает:

```
while True:  
    pass # Для прекращения работы нажмите <Ctrl+C>!
```

# Конструкция `else` цикла

Полная форма записи циклов `while` и `for` выглядит следующим образом:

```
while condition():
    operators

    # Выход с пропуском else
    if exit_test():
        break

    # Переход к заголовку цикла
    if skip_test():
        continue

# Выполняется, если не было break
else:
    operators
```

```
for x in collection:
    operators

    # Выход с пропуском else
    if exit_test():
        break

    # Переход к заголовку цикла
    if skip_test():
        continue

# Выполняется, если не было break
else:
    operators
```

Если циклы `while` или `for` прервать оператором `break`, соответствующие им блоки `else` выполнятся не будут.

## Конструкция `else` цикла

В приведенном примере выполняется проверка, является ли положительное целое число `y` простым, за счет поиска сомножителей больше 1:

```
x = y // 2 # Для y > 1

while x > 1:
    if y % x == 0: # Остаток от деления
        print(y, 'has factor', x) # Имеет сомножитель
        break # Пропуск else

    x -= 1

else: # Нормальный выход
    print(y, 'is prime') # Является простым
```

**ТОМСКИЙ  
ПОЛИТЕХНИЧЕСКИЙ  
УНИВЕРСИТЕТ**



**Благодарю за  
внимание!**

Вячеслав Алексеевич Чузлов  
к.т.н., доцент ОХИ ИШПР

