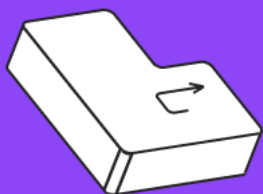




Лекция 1.

Основы Python

Погружение в Python



Оглавление

Введение	2
На этой лекции мы	2
Дополнительные материалы к лекции, которые вам пригодятся	3
Термины лекции	3
Подробный текст лекции	
1. Установка и настройка Python	
Введение	4
Установка Python	4
Работа с pip	7
Команда install	7
Команда freeze	8
Файл requirements.txt	8
Отличия командной строки ОС и интерпретатора Python	8
Работа в режиме интерпретатора	9
Использование " _".	9
Арифметические операторы в Python	9
Написание кода в интерактивном режиме	10
Задание	11
2. Повторяем основы Python	11
Работа в IDE	12
Python Style. PEP-8 (руководство по стилю) и PEP-257 (оформление документации/комментариев)	13
Переменные и требования к именам	13
Константы	15
True, False, None	15
Функция id()	15
Зарезервированные слова, keyword.kwlist	16
Ввод и вывод данных	16
Вывод, функция print()	16
Ввод, функция input()	1
Антипаттерн "магические числа"	19
Задание	19
3. Ветвление	20
Если, if	20
Отступы вместо фигурных скобок	20
Иначе, else	21

Еще если, elif	22
Выбор из вариантов, match и case	22
Логические конструкции, or, and, not	23
Ленивый if	24
Проверка на вхождение, in	24
Тернарный оператор	25
Задание	26
4. Циклы	27
Логический цикл while	27
Синтаксический сахар	27
Возврат в начало цикла, continue	28
Досрочное завершение цикла, break	29
Действие после цикла, else	29
Цикл итератор for in	31
Цикл по целым числам, он же арифметический цикл, функция range()	31
Имена переменных в цикле	32
Цикл с нумерацией элементов, функция enumerate()	32
Задание	33
5. Выводы	34

Введение

На курсе мы погрузимся в разработку на языке Python. Начнём с повторения основ, которые есть в любом языке программирования. Далее погрузимся в особенности написания программ на Python в процедурном стиле. Изучим особенности и возможности языка, такие как: функции, итераторы и генераторы, декораторы. Разберём обработку исключений и основы тестирования кода. Часть курса посвятим объектно-ориентированному программированию (ООП) на Python. В финале вас ждёт обзор стандартной библиотеки — "батареек" Python.

На этой лекции мы

1. Разберём установку и настройку Python.
2. Изучим правила создания виртуального окружения и работу с pip.
3. Повторим основы синтаксиса языка Python.
4. Познакомимся с рекомендациями по оформлению кода.
5. Изучим способы создания ветвящихся алгоритмов в Python.
6. Разберём варианты реализации циклов.
7. Узнаем про математические модули Python.

Дополнительные материалы к лекции, которые вам пригодятся

1. Инструкции по установке Python
https://drive.google.com/drive/folders/1f6J22TY_ga7ufS3Pu84rKYhskZ6Hjws2?usp=sharing
2. Википедия. Високосный год
https://ru.wikipedia.org/wiki/%D0%92%D0%B8%D1%81%D0%BE%D0%BA%D0%BE%D1%81%D0%BD%D1%8B%D0%B9_%D0%B3%D0%BE%D0%B4

Термины лекции

- **UTF-8 (от англ. Unicode Transformation Format, 8-bit — «формат преобразования Юникода, 8-бит»)** — распространённый стандарт кодирования символов, позволяющий более компактно хранить и передавать символы Юникода, используя переменное количество байт (от 1 до 4), и обеспечивающий полную обратную совместимость с 7-битной кодировкой ASCII. Кодировка UTF-8 сейчас является доминирующей в веб-пространстве. Она также нашла широкое применение в UNIX-подобных операционных системах.
- **Операционная система, сокр. ОС (англ. operating system, OS)** — комплекс управляющих и обрабатывающих программ, которые, с одной стороны, выступают как интерфейс между устройствами вычислительной системы и прикладными программами, а с другой стороны — предназначены для управления устройствами, управления вычислительными процессами,

эффективного распределения вычислительных ресурсов между вычислительными процессами и организации надёжных вычислений.

- **IDE (Integrated Development Environment)** – это интегрированная, единая среда разработки, которая используется разработчиками для создания различного программного обеспечения. IDE представляет собой комплекс из нескольких инструментов, а именно: текстового редактора, компилятора или интерпретатора, средств автоматизации сборки и отладчика.
- **PIP (Package Installer for Python)** — система управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python.
- **PEP (Python Enhancement Proposal, «Предложение по улучшению Python»)**. Это документы, предлагающие новые особенности языка. Они образуют официальную документацию особенности языка, принятие или отклонение которой обсуждается в сообществе Python.

Подробный текст лекции

1. Установка и настройка Python

Введение

Тут надо рассказать пару слов о себе на фоне слайда.

Все примеры кода в рамках этого курса будут показаны/запущены в 64-разрядной версии Python 3.10. Для успешного выполнения заданий достаточно установить Python 3.9 или новее для вашей операционной системы. Некоторые примеры могут работать иначе или вовсе не работать, если у вас установлена более ранняя версия.

Установка 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 переносимости написанных на нём программ. Язык является полностью объектно-ориентированным в том плане, что

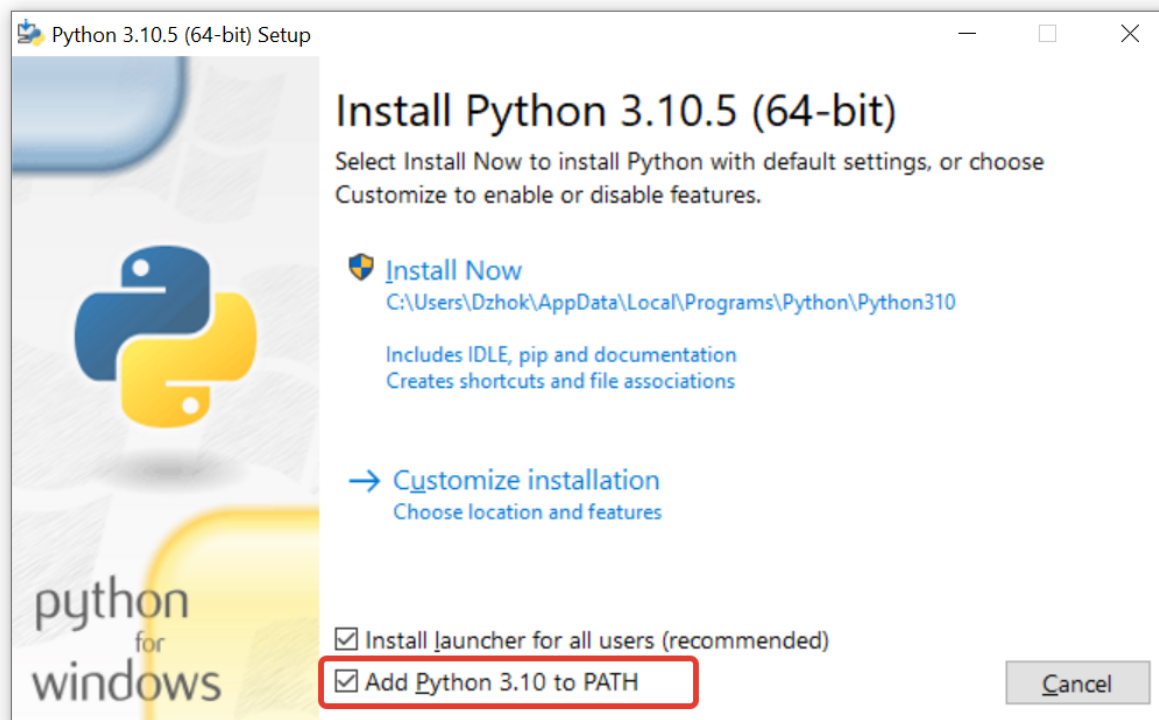
всё является объектами. О них мы будем регулярно говорить на курсе.

По умолчанию Python может быть не установлен на вашей ОС. Например, если у вас Windows. Либо у вас стоит старая версия Python, если вы используете Linux или MacOS. Чтобы гарантированно иметь нужный интерпретатор стоит установить актуальную версию Python.

- Переходим на официальный сайт Python в раздел "Загрузки" <https://www.python.org/downloads/>
- Выбираем вашу версию операционной системы.
- Скачиваем установочный дистрибутив
- Запускаем установку, отвечаем на вопросы и ждём завершения.



Важно! Если в установщике есть поле "Add Python to PATH", обязательно ставим галочку. Это упростит работу с Python из командной строки.



Чтобы убедиться в установке интерпретатора откроем командную строку вашей ОС и введем команду

```
python --version
```

В ответ ОС вернёт нам версию интерпретатора.



Внимание! В некоторых ОС команда может выдать ошибку. В таком случае пробуем более точно указать версию Python. Например `python3 --version` или `python3.10 --version`

Если у вас возникли сложности с установкой, обратитесь к инструкции по установке для вашей ОС. Они приложены в материалы к уроку.

Создание виртуального окружения

Запуск простых программ возможен непосредственно из ОС. Но когда программист работает над большим проектом, используется виртуальное окружения. Так мы изолируем проект от других, хранящихся на этом же ПК. Кроме того изменения внутри проекта не влияют на "эталонную" версию Python, которую только что установили.

Для начала создайте каталог для нового проекта.



Важно! Не все устанавливаемые модули Python дружат с кириллицей, пробелами и вообще любыми символами помимо латиницы. Путь от корня диска до каталога должен быть без пробелов, а названия директорий написаны латинскими буквами. Допускается символ подчеркивания.

```
Выбрать dzhoker@DESKTOP-DI3TAQF: /mnt/c/Users/Dzhok/new_project
dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok$ mkdir new_project
dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok$ cd new_project/
dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok/new_project$
```

После создания каталога перейдите в него.

Далее создаём копию Python внутри рабочего каталога одной из команд (Windows или Unix)

```
python -m venv venv - для Windows;
python3 -m venv venv - для Linux и MacOS.
```

Будет создана папка `venv`. Её мы указали в конце команды. Первый `venv` - вызов модуля для создания окружения. В указанную папку копируется локальная версия интерпретатора.

Остаётся активировать виртуальное окружение. Это финальная команда, которая отличается в Windows и Unix системах.

```
venv\Scripts\activate - для Windows;  
source venv/bin/activate - для Linux и MacOS.
```

Если в начале командной строки вы видите приставку (`venv`), значит активация прошла успешно. Можем работать над проектом в изолированной среде.

Сразу уточню, что для завершения работы внутри виртуального окружения необходимо выполнить команду

```
deactivate
```



Важно! Виртуальное окружение созданное в одной ОС нельзя скопировать в другую ОС. Мультиплатформенность работает с кодом на языке Python. А виртуальные окружения, IDE и т.п. индивидуальны для каждой ОС и независимо устанавливаются в начала процесса разработки.

Работа с `pip`

Package Installer for Python — система управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python. Благодаря `pip` мы можем устанавливать сторонние библиотеки, фреймворки, пакеты. Например через `pip` устанавливаются Django, Flask, NumPy.



Важно! Прежде чем устанавливать дополнения убедитесь, что вы находитесь в каталоге проекта и активировали виртуальное окружение.

➤ Команда `install`

Для установки используем команду `install` далее указываем название устанавливаемого дополнения.

Потренируемся с установкой и добавим в проект библиотеку `requests`. `Requests` — это простая, но элегантная библиотека для работы с HTTP.

```
pip install requests
```


"Пип" скачает и установит библиотеку requests последней версии в папку venv виртуального окружения. В основной системе и в других окружениях библиотеки не будет.

➤ Команда freeze

Убедимся, что установка прошла успешно. Выполним команду

```
pip freeze
```

Мы получили список всех дополнений внутри нашего виртуального окружения. Помимо requests были скачаны библиотеки зависимости, необходимые для работы requests. Данный процесс автоматизирован. Зависимости указывает разработчик, а pip рекурсивно их устанавливает.

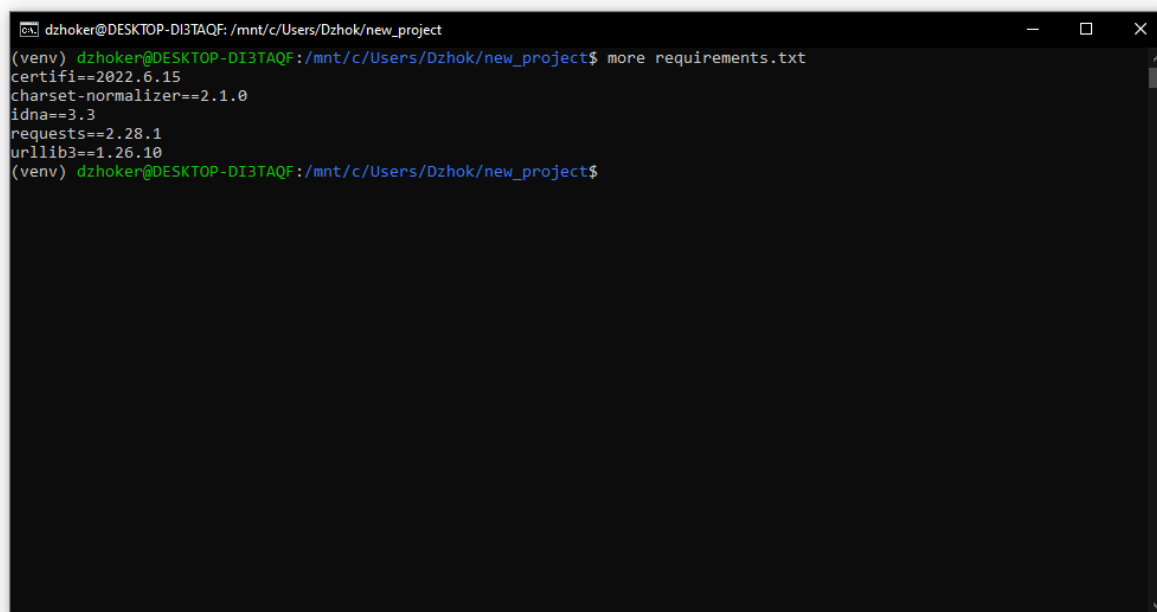
➤ Файл requirements.txt

Чтобы в будущем облегчить развертывание проекта на новом/боевом/тестовом сервере, используют файл requirements.txt. В него помещают перечень всех уже установленных дополнений. Для этого используют команду

```
pip freeze > requirements.txt
```

Откроем файл и посмотрим на его содержимое.

more requirements.txt

A screenshot of a terminal window with a black background and white text. The window title is 'dzhoker@DESKTOP-DI3TAQF: /mnt/c/Users/Dzhok/new_project'. The command '(venv) dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok/new_project\$ more requirements.txt' has been executed. The output lists the installed packages and their versions: 'certifi==2022.6.15', 'charset-normalizer==2.1.0', 'idna==3.3', 'requests==2.28.1', and 'urllib3==1.26.10'. The prompt '(venv) dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok/new_project\$' is visible at the bottom.

```
dzhoker@DESKTOP-DI3TAQF: /mnt/c/Users/Dzhok/new_project
(venv) dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok/new_project$ more requirements.txt
certifi==2022.6.15
charset-normalizer==2.1.0
idna==3.3
requests==2.28.1
urllib3==1.26.10
(venv) dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok/new_project$
```

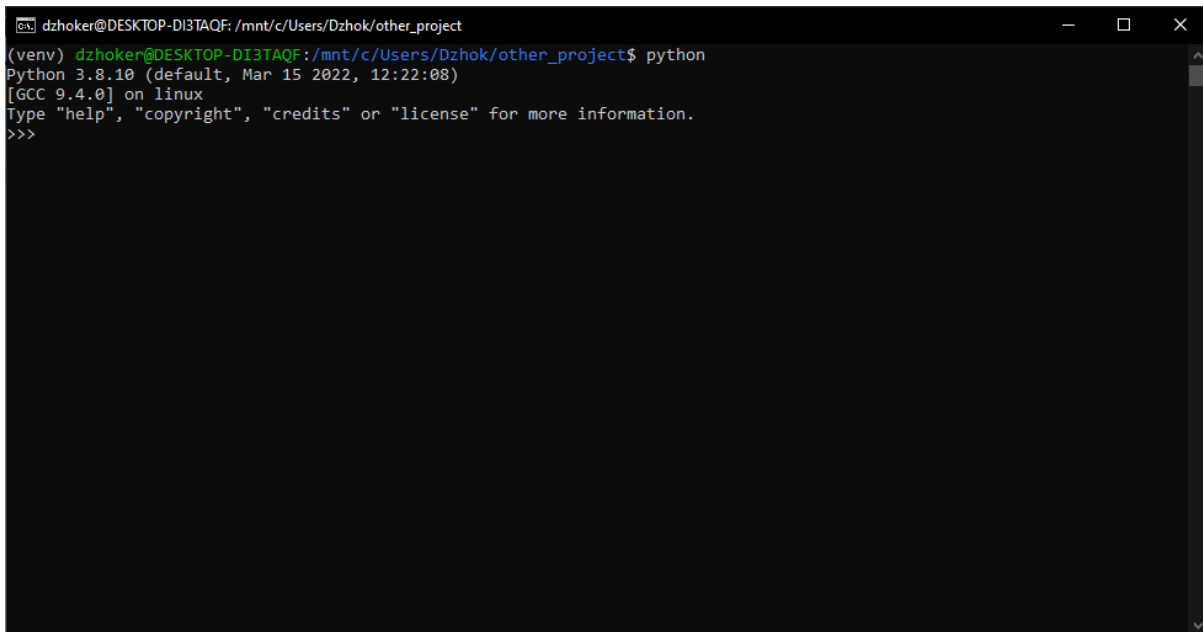
Для быстрой установки в новое окружение всего содержимого файла используется команда

```
pip install -r requirements.txt
```

➤ Отличия командной строки ОС и интерпретатора Python

Всё, что мы делали до этого делалось в командной строке операционной системы. Мы передавали ОС команды и дополнительные параметры и получали результат. Речь шла о Python, но команды получал и выполнял не он, а ОС.

Пришло время написать свою первую программу. И писать её мы будем в режиме интерпретатора. Выполним команду `python` (либо `python3`) чтобы активировать интерпретатор

A screenshot of a terminal window with a dark background. The title bar at the top shows the user 'dzhoker' and the directory '/mnt/c/Users/Dzhok/other_project'. The terminal text shows the command 'python' being executed, which starts the Python 3.8.10 interpreter. It displays version information and the GCC version, then prompts the user with '>>>' for input.

```
dzhoker@DESKTOP-DI3TAQF: /mnt/c/Users/Dzhok/other_project
(venv) dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok/other_project$ python
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Обратите внимание на три треугольные стрелки. Теперь мы находимся в интерактивном сеансе Python. В нём не будут работать команды ОС, такие как `cd`, `pip` и другие.

Работа в режиме интерпретатора

Особенностью работы в режиме интерпретатора является то, что каждая строка кода выполняется сразу после нажатия клавиши Ввод. Есть и исключения, но о них через минуту. А пока воспользуемся оболочкой как калькулятором. Можно вводить любые математические операции и сразу получать ответ.

Использование `"_"`.

А если нам нужен последний вывод, можно воспользоваться переменной `"_"`. Она хранит именно финальный результат, т.е. обновляется с каждым новым выводом.

```
dzhoker@DESKTOP-DI3TAQF: /mnt/c/Users/Dzhok/other_project
(venv) dzhoker@DESKTOP-DI3TAQF:/mnt/c/Users/Dzhok/other_project$ python
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2+2
4
>>> 8*6
48
>>> 4+3*8-(2+4)
22
>>> _ + 2
24
>>>
```

➤ Арифметические операторы в Python

Основные математические операторы представлены в таблице.

Оператор	Описание	Примеры
+	Сложение	$398 + 20 = 418$
-	Вычитание	$200 - 50 = 150$
*	Умножение	$34 * 7 = 238$
/	Деление	$36 / 6 = 6.0$ $36 / 5 = 7.2$
//	Целочисленное деление	$36 // 6 = 6$ $9 // 4 = 2$ $-9 // 4 = -3$ $15 // -2 = -3$
%	Остаток от деления	$36 \% 6 = 0$ $36 \% 5 = 1$
**	Возведение в степень	$2 ** 16 = 65536$

Обратите внимание на операцию целочисленного деления с участием отрицательных чисел в качестве делимого или делителя. Такой результат

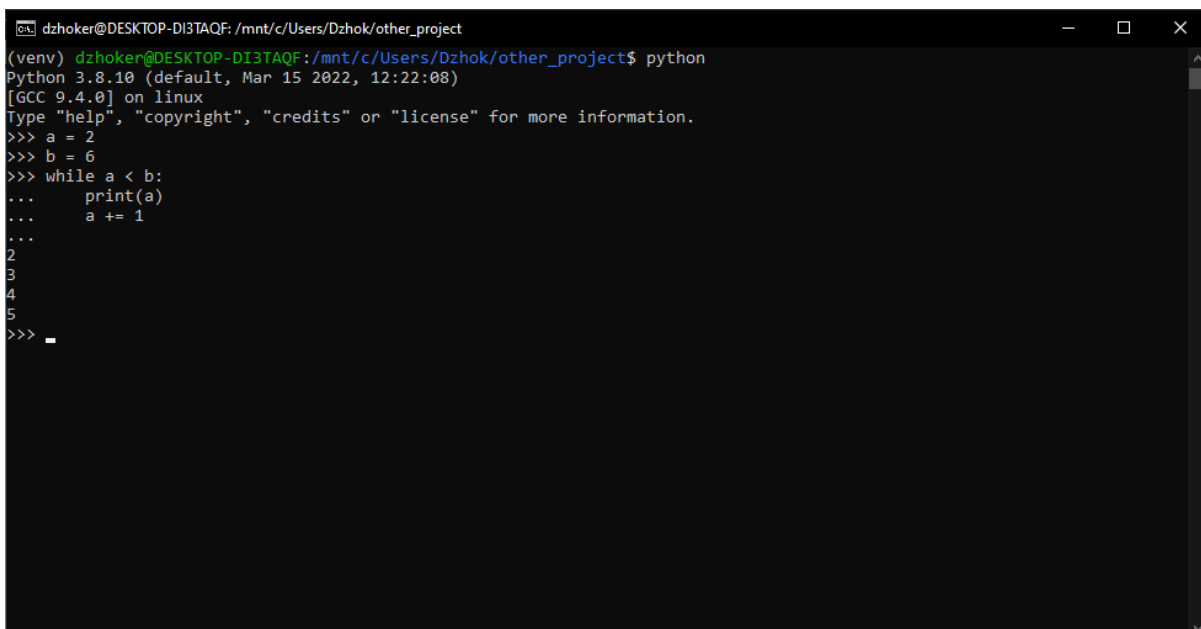
обусловлен тем, что целочисленное деление в Python округляет итоговое значение в меньшую сторону.

```
9 / 4 = 2.25, округляем до меньшего целого - это 2
-9 / 4 = -2.25, округляем до меньшего целого - это -3
```

➤ Написание кода в интерактивном режиме

Мы можем писать программы любой сложности, использовать переменные, ветвления, циклы внутри оболочки. При этом наличие двоеточия позволяет писать несколько строк кода без вывода результата. Python запоминает целый блок и выводит ответ после нажатия клавиши Ввод дважды.

```
>>> a = 2
>>> b = 6
>>> while a < b:
...     print(a)
...     a += 1
...
2
3
4
5
```



```
dzhoker@DESKTOP-DI3TAQF: /mnt/c/Users/Dzhok/other_project
(venv) dzhoker@DESKTOP-DI3TAQF: /mnt/c/Users/Dzhok/other_project$ python
Python 3.8.10 (default, Mar 15 2022, 12:22:08)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 2
>>> b = 6
>>> while a < b:
...     print(a)
...     a += 1
...
2
3
4
5
>>> .
```

У оболочки есть один минус. После выхода мы теряем все введённые данные. Поэтому для написания большого кода используются файлы с расширением ru. Далее по курсу мы будем использовать их.

А пока завершим интерактивный сеанс командой

```
exit()
```

Задание

Проверим усвоение материала. Вот несколько простых вопросов. Для ответа на каждый вам даётся одна минута. Ответы напишите в чат.

1. Какая версия Python у вас установлена? Если не установлена, какую версию будете устанавливать?
2. Запомнили ли вы имя файла, в котором сохраняется перечень всех необходимых дополнений для вашего проекта. Как он называется?
3. Как получить доступ к результату последней операции в режиме интерпретатора Python?

2. Повторяем основы Python

Пришло время поговорить о базовых вещах в программировании применительно к языку Python.

Работа в IDE

Python-программы можно писать в любом текстовом редакторе, главное — сохранять текст программы в кодировке UTF-8. Свой код буду демонстрировать в IDE PyCharm. Версия Community бесплатна. Код сохраняется в файлах с расширением `py` и может быть многократно использован. Однако вы можете писать код в любой другой IDE, к которой привыкли. Стоит вспомнить, что Python является мультиплатформенным языком программирования, а значит разные операционные системы будут одинаково выполнять код. Аналогично и IDE являются лишь оберткой, которая помогает писать код разработчику, но никак не влияет на результаты выполнения Python программ. Используйте ту IDE в которой вам удобно работать.



Важно! Не давайте именам питоновских файлов имена встроенных и внешних модулей. Это приведёт к затиранию имён и отключению функционала модулей.

Python Style. PEP-8 (руководство по стилю) и PEP-257 (оформление документации/комментариев)

Одно из важнейших требований к коду Python-разработчика — следование стандарту PEP-8, который представляет собой описание рекомендованного стиля кода. Причем PEP-8 действует для основного текста программы, а для строк документации разработчику рекомендуется придерживаться положений PEP-257. Документ содержит достаточно объёмное описание стандарта. В процессе этого курса мы будем знакомиться с различными пунктами из этих "пеп". Привыкайте к верному стилю с самого начала. Ведь фразу "Превед, каг тваи дила" прочитать можно. Но доверять такому автору написание шедевра уже не хочется. Далее на протяжении курса буду упоминать о правилах по написанию читаемого кода, уточняя что это "пеп8".

Переменные и требования к именам

В Python всё объект. Числа, строки, массивы и даже функции и классы являются объектом. Переменная в Python является указателем на объект. Разберём на примере.

```
a = 5
```

Переменная "a" хранит значение "5". Верно. Также верно, что Python создал объект целого типа 5 (о типах данных мы поговорим на следующих лекциях). Далее была создана переменная "a" которая является указателем на объект целого типа число пять.

Python является языком со строгой динамической типизацией. Это означает что тип объекта изменить невозможно он строго задаётся при создании объекта. При этом переменные могут ссылаться на объекты разных типов. И это не вызывает ошибки.

```
a = 5
a = "hello world"
a = 42.0 * 3.141592 / 2.71828
```



PEP-8! Обратите внимание на отступы в один пробел вокруг математических знаков "=", "*" и др. Это требование по оформлению кода, которое повышает его читаемость.



Важно! Использование одной и той же переменной для разных типов данных является плохим стилем. Предварительная инициализация переменных в языке не нужна. Вы можете использовать переменную в том месте где оно понадобилось. Даже если уже написали несколько сотен строк кода.

Что ещё нужно знать когда речь идёт о переменных и их именах:

- Python использует кодировку utf-8. Поэтому в качестве имени переменной может выступать любой набор символов, даже смайлик. Однако правильные имена это имена на латинице, т.е. английские строчные буквы, слова. Например name, age.
- Если название переменной состоит из нескольких слов такие слова записываются строчными буквами разделяются символом подчёркивания. Этот стиль называется snake_case.
- В качестве первого символа в имени переменной запрещено использовать цифры и другие знаки пунктуации в этом случае Python выдаст ошибку. Имя начинается с буквы или символа подчёркивания.
- Не используйте написание слов на транслите. Воспользуйтесь онлайн переводчиком на английский. Не zdorove, а health.

Верно	Не верно
first_name, user_1, request, _tmp_name, min_step_shift	1_name, User_1, 0request, tmpName, minStep_shift, 😊, имя

Константы

Дополнительных команд для создания констант в языке Python нет. Есть лишь договорённость что константа — это переменная написанное прописными буквами.

Примеры констант.

```
MAX_COUNT = 1000
ZERO = 0
DATA_AFTER_DELETE = 'No data'
DAY = 60 * 60 * 24
```

Константа в программировании — способ адресации данных, изменение которых программой не предполагается или запрещается. Python не вызовет исключение, если вы измените константу внутри кода. Но подобные действия со стороны программиста являются неверными.

True, False, None

Отдельно хочу выделить три встроенные в язык Python константы. Это истина True, ложь False и ничего None. Первая буква строчная остальные прописные. Истину или ложь мы получаем в результате логических операций. О них мы поговорим чуть позже. Значение None означает "ничего". Его можно использовать для создания переменной, значение которой изначально мы не знаем. Также None возвращают некоторые функции, результат работы которых не подразумевает возврат значения.

Функция id()

Вернёмся к тому что в Python всё объект, а переменная является ссылкой на объект. Воспользуемся встроенной функцией id(), которая возвращает адрес объекта в оперативной памяти вашего компьютера.

```
a = 5
print(id(a))
a = "hello world"
print(id(a))
a = 42.0 * 3.141592 / 2.71828
print(id(a))
```

В отличие от режима интерпретатора, запуск программных файлов не выводит значения на экран. Поэтому мы используем функцию print(), внутри которой передаём вызов функции id(a). Как вы видите одна и та же переменная возвращает три разных адреса для трёх разных объектов в памяти Python.


```
1615821406576
1615826697776
1615831485936
```

Зарезервированные слова, keyword.kwlist

Существует чуть менее 40 зарезервированных слов, которые образуют базовый синтаксис языка Python. Ниже представлены они все.

```
False, None, True, 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.
```

А также case и match начиная с версии Python 3.10.

Первую Троицу из списка мы разобрали, когда говорили о встроенных в Python константах. О остальных зарезервированных словах мы будем говорить далее на протяжении всего курса. Уверен, что некоторые слова вам знакомы по прошлым курсам и другим языкам программирования.



Важно! Запрещено использовать в качестве имён переменных зарезервированные слова. Python завершит код с ошибкой.

Ввод и вывод данных

Для ввода и вывода данных в консоль поэтому используются стандартные потоки ввода-вывода. Однако программист не работает с ними напрямую, а использует встроенные в язык функции `print()` для вывода данных и `input()` для ввода данных и сохранение в переменные.

➤ Вывод, функция `print()`

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Функция `print()` принимает один или несколько объектов разделённых запятыми и выводит их на печать. По умолчанию это вывод в консоль, т.е. стандартный поток вывода.

```
print(42)
print(1, 2, 3, 4)
print('Hello', ',', 'world', '!')
```

Вывод:

```
42
1 2 3 4
Hello , world !
```

Отдельно уточню про два ключевых аргумента `sep` и `end`.

`sep` по умолчанию хранит один пробел. Именно этим символом разделяются все объекты, перечисленные через запятую.

`end` по умолчанию хранит символ перехода на новую строку `'\n'`. Это то, что функция `print` добавляет после вывода всех объектов.

```
print(42, sep='___', end='\n(=^.^=)\n')
print(1, 2, 3, 4, sep='___', end='\n(=^.^=)\n')
print('Hello', ',', 'world', '!', sep='___', end='\n(=^.^=)\n')
```

Вывод:

```
42
(=^.^=)
1___2___3___4
(=^.^=)
Hello___,___world___!
(=^.^=)
```

Обратите внимание, что текст, заключённый в одинарные или двойные кавычки, выводится без изменения. Python воспринимает его как строковую информацию. Для вывода содержимого переменных мы указываем имя переменной без кавычек.

```
number = 42
print(number, sep='___', end='\n(=^.^=)\n')
ONE = 1
TWO = 2
print(ONE, TWO, 3, 4, sep='> <', end='>')
```

Вывод:

```
42
```

```
(=^ . ^=)
1> <2> <3> <4>
```

➤ Ввод, функция input()

Для ввода данных и сохранение их в переменной используется функция input().

```
result = input([prompt])
```

Если в качестве аргумента функции передать значение, оно будет выведено как подсказка перед вводом данных. Функция возвращает объект строкового типа, который можно сохранить в переменную.

```
name = input('Ваше имя: ')
```

Обратите внимание на следующий пример.

```
age = input('Ваш возраст: ')
```

В переменной age будет сохранено значение в строковом формате. Посчитать сколько лет назад человеку стукнуло 18 не получится.

```
how_old = age - 18
TypeError: unsupported operand type(s) for -: 'str' and 'int'
```

Для приведения строки к числу используем функции int() — целый тип или float() — вещественный тип, число с плавающей запятой. Подробнее о них поговорим на следующей лекции.

```
age = float(input('Ваш возраст: '))
how_old = age - 18
print(how_old, "лет назад ты стал совершеннолетним")
```

Вывод:

```
Ваш возраст: 33
15.0 лет назад ты стал совершеннолетним
```



Важно! Если попытаться преобразовать к числу не число, получим ошибку.

```
ValueError: could not convert string to float: 'пять'
```

Способы обработки пользовательского ввода и обработка ошибок — тема одной из следующих лекций. Пока договоримся об идеальном пользователе, который вводит числа там где это необходимо и не создаёт ошибок.

Антипаттерн "магические числа"

В прошлом примере мы использовали антипаттерн — плохой стиль для написания кода. Число 18 используется в коде без пояснений. Такой антипаттерн называется "магическое число". Рекомендуется помещать числа в константы, которые хранятся в начале файла.

```
ADULT = 18
age = float(input('Ваш возраст: '))
how_old = age - ADULT
print(how_old, "лет назад ты стал совершеннолетним")
```

Плюсом такого подхода является возможность легко корректировать большие проекты. Представьте, что в вашем коде несколько тысяч строк, а число 18 использовалось несколько десятков раз.

- При развертывании проекта в стране, где совершеннолетием считается 21 год вы будете перечитывать весь код в поисках магических "18" и править их на "21". В случае с константой изменить число нужно в одном месте.
- Дополнительный сложности могут возникнуть, если в коде будет 18 как возраст совершеннолетия и 18 как коэффициент для расчёт чего-либо. Теперь править кода ещё сложнее, ведь возраст изменился, а коэффициент -нет. В случае с сохранением значений в константы мы снова меняем число в одном месте.

Задание

Очередная серия из трёх вопросов. По две минуты на ответ. Его вы пишете в чат.

1. Какое из слов лишнее и что означают остальные слова: True, NaN, False, None?
2. Какие имена переменных плохие, а какие правильные и почему: name, 1_year, _hello, id, MAX_DATE, zdorove?
3. Что вы можете рассказать про магические числа?

3. Ветвление

Ветвление в программировании — операция, применяющаяся в случаях, когда выполнение или невыполнение некоторого набора команд должно зависеть от выполнения или невыполнения некоторого условия. Ветвление — одна из трёх базовых конструкций структурного программирования.

"Некоторое условие" должно возвращать истину True или ложь False. В Python есть возможность использовать неявное преобразование типов. Так любое целое число помимо нуля считается истиной, а ноль — ложью. Любая коллекция с данными — истина, а пустая коллекция — ложь. Подробнее хорошие приёмы неявного преобразования мы рассмотрим далее на курсе. А пока стоит запомнить, что явное лучше неявного.

Если, if

Простейшая проверка условия происходит при помощи зарезервированного слова if

```
pwd = 'text'
res = input('Input password: ')
if res == pwd:
    print('Доступ разрешён')
```

В строке 3 мы используем if, далее пишем выражение, которое должно вернуть истину или ложь и завершаем строку двоеточием.

В Python доступны все 6 операций сравнения:

«==» — равно	«!=» — не равно
«>» — больше	«<=» — меньше или равно
«<» — меньше	«>=» — больше или равно

Отступы вместо фигурных скобок

Обратите внимание на отступ в четыре пробела в следующей после if строке кода. В отличие от Си-подобных языков программирования? таких как Java, C# и других, Python не используют фигурные скобки для создания логических блоков кода.

Вместо этого мы создаём отступы из четырёх пробелов. Такой стиль повышает читаемость кода и сокращает количество ошибок.

```
pwd = 'text'
res = input('Input password: ')
if res == pwd:
    print('Доступ разрешён')
    print('Но будьте осторожны')
print('Работа завершена')
```

Строки 4-5 написаны с отступом и будут выполнены в случае истинности в строке 3. Строка 6 будет выполнена в любом случае, т.к. написана без отступов.



PEP-8! рекомендует использовать в качестве отступа 4 пробела.

Программа будет верно работать и с 2 и с 8 отступами, и даже с символом табуляции (при условии что во всём коде использован единый стиль отступания). Но программы проверки кода — линтеры будут выдавать предупреждения, а коллеги по проекту подзатыльники.

Также обратите внимание, что IDE при верной настройке заменяют нажатие клавиши TAB на клавиатуре на ввод 4 пробелов.

Обратите внимание на отсутствие скобок в строке 3. Python не требует заключать логическое выражение в скобки. А IDE с проверкой синтаксиса будет ругаться на скобки, как на рудимент. Единственный случай, когда вам могут понадобиться скобки, построение сложных логических выражений, где надо поменять порядок логических проверок с традиционного слева направо, на другой.

Иначе, else

Для выполнения кода в случае ложности логического выражения используется зарезервированное слово `else` с обязательным двоеточием после него.

```
pwd = 'text'
res = input('Input password: ')
if res == pwd:
    print('Доступ разрешён')
    print('Но будьте осторожны')
else:
    print('Доступ запрещён')
print('Работа завершена')
```

Слово `else` относится к тому `if`, с которым находится на одном уровне. В примере ниже верхний `if` связан с нижним `else`, а средний `if` со средним `else`.

```
pwd = 'text'
res = input('Input password: ')
if res == pwd:
    print('Доступ разрешён')
    my_math = int(input('2 + 2 = '))
    if 2 + 2 == my_math:
        print('Вы в нормальном мире')
    else:
        print('Но будьте осторожны')
else:
    print('Доступ запрещён')
print('Работа завершена')
```

Еще если, `elif`

Для проверки нескольких выражений используется `elif` - сокращение от `else if`.

```
color = input('Твой любимый цвет: ')
if color == 'красный':
    print('Любитель яркого')
elif color == 'зелёный':
    print('Ты не охотник?')
elif color == 'синий':
    print('Ха, классика!')
else:
    print('Тебя не понять')
```

Проверка работает до первого совпадения. После него дальнейший код пропускается. А если совпадений нет, срабатывает код после `else`.

Выбор из вариантов, `match` и `case`

В Python версии 3.10, т.е. совсем недавно появилась новая возможность множественного сравнения. Это конструкция `match` и `case`. После `match` указываем

переменную для сравнения. Далее идёт блок из множества case с вариантами сравнения. Рассмотрим работу кода на примере.



Важно! Если у вас стоит Python версии 3.9 и ниже, код не будет работать.

```
color = input('Твой любимый цвет: ')
match color:
    case 'красный' | 'оранжевый':
        print('Любитель яркого')
    case 'зелёный':
        print('Ты не охотник?')
    case 'синий' | 'голубой':
        print('Ха, классика!')
    case _:
        print('Тебя не понять')
```

Данный код аналогичен прошлому варианту с elif. Добавлена возможность проверить несколько цветов. Например для красного и оранжевого будет один вывод. Вертикальная черта играет роль оператора "или". Уточню что пользователь вводит один единственный цвет.

Вместо слова else в данной конструкции используется сочетание case _ На этом курсе мы ещё несколько раз будет встречаться с подчеркиванием. И каждый раз он имеет разные эффект в зависимости от применения.

Логические конструкции, or, and, not

В Python доступны три логических оператора:

- and — логическое умножение «И»;
- or — логическое сложение «ИЛИ»;
- not — логическое отрицание «НЕ».

Логика их работы представлена в таблице

first	second	first and second	first or second	not first
True	True	True	True	False
False	True	False	True	True
True	False	False	True	-

False	False	False	False	-
-------	-------	-------	-------	---

А теперь пример кода на Python чтобы разобраться в правильном синтаксисе построения логических выражений. Вычислим високосный год в Григорианском календаре поэтапно:

```
year = int(input('Введите год в формате yyyy: '))
if year % 4 != 0:
    print("ОБЫЧНЫЙ")
elif year % 100 == 0:
    if year % 400 == 0:
        print("Високосный")
    else:
        print("ОБЫЧНЫЙ")
else:
    print("Високосный")
```

А теперь выберем все случаи, когда год обычный и запишем их в одну строку:

```
if year % 4 != 0 or year % 100 == 0 and year % 400 != 0:
    print("ОБЫЧНЫЙ")
else:
    print("Високосный")
```

Python последовательно слева направо проверяет логическое выражение, формируя финальный ответ — True или False.

Ленивый if

Ещё раз посмотрим на прошлый пример кода.

```
if year % 4 != 0 or year % 100 == 0 and year % 400 != 0:
```

В Python как и в некоторых других языках программирования if "ленивый". Если в логическом выражении есть оператор `or` и первое значение то есть левое вернуло истину, дальнейшая проверка не происходит, возвращается True. Если в логическом выражении есть оператор `and` и левая половина вернула ложь, то возвращается False без проверки правой половины выражения.

Проверка на вхождение, in

На одном из следующих уроках поговорим о коллекциях. Пока для простоты договоримся, что список целых чисел в квадратных скобках — массив данных. Оператор `in` проверяет вхождение элемента в последовательность.

```
data = [0, 1, 1, 2, 3, 5, 8, 13, 21]
num = int(input('Введи число: '))
if num in data:
    print('Леонардо передаёт привет!')
```

А теперь тот же самый код, но с конструкцией `not` — отрицание:

```
data = [0, 1, 1, 2, 3, 5, 8, 13, 21]
num = int(input('Введи число: '))
if num not in data:
    print('Леонардо грустит :-(')
```

Обратите внимание на порядок слов в строке 3. Python использует максимально приближенные к человеческому, а точнее к английскому языку стиль программирования: "Если число не входит в данные, то..."

Тернарный оператор

Завершим разговор о ветвлениях тернарным оператором. Он позволяет записать 4 логические строки в одну.

Было:

```
my_math = int(input('2 + 2 = '))
if 2 + 2 == my_math:
    print('Верно!')
else:
    print('Вы уверены?')
```

Стало:

```
my_math = int(input('2 + 2 = '))
print('Верно!' if 2 + 2 == my_math else 'Вы уверены?')
```

Слева от if записываем выражение для истины, справа от else результат для лжи.



Важно! При использовании тернарного оператора важно помнить, что код должен сохранить свою читаемость. Слишком длинные и сложные конструкции стоит переписать в более привычном виде в 4 строки.



PEP-8! требует, чтобы длина строки кода не превышала 80 символов. Современные мониторы с широкими экранами и высоким разрешением с легкостью вмещают больше символов. Поэтому ограничение было пересмотрено и увеличено до 120 символов. Более длинные строки кода в Python недопустимы, т.к. снижают его читаемость.

Задание

1. Перед вами пример кода. Значения переменных `num` и `name` указаны в первых строчках. Попробуйте прочитать код и выбрать верный вариант без запуска кода. У вас две минуты на каждую пару значений.

```
num = 42
name = 'Bob'
if num > 30:
    if num < 50:
        print('Вариант 1')
    elif name > 'Markus':
        print('Вариант 2')
    else:
        print('Вариант 3')
elif name < 'Markus':
    print('Вариант 4')
elif num != 42:
    print('Вариант 5')
else:
    print('Вариант 6')
```

2. А что получится теперь:

```
num = 64
```

```
name = 'Bob'
```

3. И финальные переменные. Какой вариант будет теперь:

```
num = 7  
name = 'Neo'
```

4. Циклы

Рассмотрим способы создания циклов в Python.



Важно! Мы снова будем использовать отступы в 4 пробела. Это неотъемлемая часть синтаксиса языка.

Логический цикл while

Цикл `while` является циклом с предусловием. Проверяем логическое условие, аналогично "если" и в случае истинности выполняем вложенный блок кода. Далее возвращаемся к проверке условия.

Попробуем перебрать всё чётные числа от нуля до введённого пользователем исключительно..

```
num = float(input('Введите число: '))  
count = 0  
while count < num:  
    print(count)  
    count += 2
```

Проверка условия после `while` и выполнение тела цикла продолжается до тех пор, пока условие истинно.

Синтаксический сахар

Обратите внимание на нижнюю строку кода. `"+="` - синтаксический сахар Python.

Синтаксический сахар (англ. syntactic sugar) в языке программирования — это синтаксические возможности, применение которых не влияет на поведение программы, но делает использование языка более удобным для человека.

Для неизменяемых типов данных, а числа в Python неизменяемы, две следующие строки кода эквивалентны.

```
num = num + 1
num += 1
num++ # не работает в Python
```

Аналогично можно записать коротко вычитание, умножение, целочисленное деление и другие операции.

Возврат в начало цикла, continue

При необходимости работу цикла можно прервать и досрочно вернуться к проверке условия. Для этого используем зарезервированное слово `continue`.

Выведем все чётные числа (как в прошлом примере), кроме тех, которые кратны 12.

```
num = float(input('Введите число: '))
STEP = 2
limit = num - STEP
count = -STEP
while count < limit:
    count += STEP
    if count % 12 == 0:
        continue
    print(count)
```

`if` внутри цикла проверяет кратность двенадцати. В случае истинности команда `continue` возвращает нас к началу цикла, к `while`.

И пара слов о двух оптимизациях в коде:

1. `STEP = 2` — добавили константу для движения по чётным и ушли от "магических чисел". Теперь изменение условия на "вывод чисел кратных 5" потребует замены числа в одном месте кода.
2. Ввели переменную `limit`. Чтобы цикл не выводил лишние числа, больше введенного `num`, на каждой проверке цикла `while` надо вычитать значение шага. Но шаг — константа. Для экономии ресурсов ПК и ускорения работы кода логично сделать вычитание один раз перед циклом и сравнивать значения быстрее, без вычитания в строке `while`

Досрочное завершение цикла, break

Ещё один способ управления циклом — команда `break` для его досрочного завершения. Она отлично подходит для создания циклов с постусловием, бесконечных циклов с возможностью выхода.

Рассмотрим на примере программы, которая просит ввести число внутри заданного диапазона.

```
min_limit = 0
max_limit = 10
while True:
    print('Введи число между', min_limit, 'и', max_limit, '? ')
    num = float(input())
    if num < min_limit or num > max_limit:
        print('Неверно')
    else:
        break
print('Было введено число ' + str(num))
```

Конструкция `while True:` создаёт бесконечный цикл. Вместо `True` можно было бы подставить любое выражение, которое всегда возвращает истину. Но именно такая реализация обеспечивает лучшую читаемость и быстродействие.

Действие после цикла, else

Зарезервированное слово `else` может применяться не только к ветвлениям, но и к циклам. Для этого `else` должно быть расположено на том же уровне, т.е. иметь столько же пробельных отступов, что и начало цикла — `while`.

Доработаем прошлую программу и дадим 3 попытки на попадание в диапазон.

```
min_limit = 0
max_limit = 10
count = 3

while count > 0:
    print('Попытка ' + str(count))
    count -= 1
```

```

    num = float(input('Введи число между ' + str(min_limit) + ' и '
    + str(max_limit) + ': '))
    if num < min_limit or num > max_limit:
        print('Неверно')
    else:
        break

else:
    print('Исчерпаны все попытки. Сожалеею.')
    quit()

print('Было введено число ' + str(num))

```

Что изменилось в коде:

- Добавили счётчик count. Цикл проверяет не уменьшился ли счётчик до нуля. Если нет, переходим в тело цикла. Выводим значение счётчика на каждом витке цикла и уменьшает его на единицу.
- Добавили else для while. Логика следующая. Если пользователь ввёл верное число и сработала команда break, блок else для цикла игнорируется. А если числа вводились ошибочно, счётчик досчитал до нуля и произошел выход из цикла по "лжи" в while. В этом случае сработает блок кода после else для цикла.
Если коротко, вызов break в цикле игнорирует else для цикла.
- quit() — ещё одна функция для завершения кода раньше, чем мы дойдём до конца файла. Это вторая функция. Первой, exit() мы пользовались для завершения работы сеанса интерпретатора. Работают они аналогично.
- Мы добавили пустые строки в код. Python игнорирует такие строки. Но разделение кода на блоки по смыслу упрощает чтение. Кроме того, на следующих лекциях мы поговорим о пустых строках, которые прописаны в PEP-8. Но не ставьте пустые строки после каждой строки кода. Если вам нравится большой межстрочный интервал, настройте его в своей IDE.
- **PEP-8!** Кстати, последняя строка в файле должны быть пустой. Именно одна, а не ноль, и не две.

Цикл итератор for in

Цикл for in используется Python разработчиками намного чаще. Прежде чем приступить к его рассмотрению уточню, что команды continue, break и else могут применяться к циклу for in точно так же как и до этого в цикле while.

Рассмотрим работу цикла в качестве итератора последовательности. Итерироваться будем по массиву с числами из части про ветвления и проверку на вхождение.

```
data = [0, 1, 1, 2, 3, 5, 8, 13, 21]
for item in data:
    print(item)
```

Цикл последовательно перебирает все элементы массива data и поочерёдно помещает их в переменную item. После for указываем переменную или переменные для приема значений, они изменяются на каждом витке цикла. После in указываем объект, из которого последовательно берём данные.



Важно! Нельзя изменять содержимое контейнера (в нашем примере data) во время итерации по нему, т.е. внутри цикла. Это приведет к неожиданным ошибкам.

Цикл по целым числам, он же арифметический цикл, функция range()

Для перебора целых чисел цикл for in используется в связке с функцией range(). Она выступает в качестве объекта итератора. Пример печати чётных чисел от нуля до введённого числа может выглядеть так с циклом for:

```
num = int(input('Введите число: '))
for i in range(0, num, 2):
    print(i)
```



Важно! Аргументами функции range() должны быть целые числа, int()

Рассмотрим подробнее варианты работы функции range(). Обратите внимание на количество переданных функции аргументов

range(stop) — перебираем значения от нуля до stop исключительно с шагом один

range(start, stop) — перебираем значения от start включительно до stop исключительно с шагом один

range(start, stop, step) — перебираем значения от start включительно до stop исключительно с шагом step.

Пара если.

- Если значение step отрицательное, перебор будет в сторону уменьшения.

- Если start больше stop при положительном step или наоборот start меньше stop при отрицательном step, цикл не сработает ни разу. `range(10, 5, 2)` - ничего

Имена переменных в цикле

Исторически сложилось, что для переменных, которые "считают арифметику" в цикле используются переменные `i`, `j`, `k`. Именно в таком порядке с учётом вложенности. Этой традиции более полувека. Так что смело продолжайте её и ваш код поймут. Например так выглядят 3 вложенных цикла. Обратите внимание на отступы, которые показывают уровень вложенности

```
count = 10
for i in range(count):
    for j in range(count):
        for k in range(count):
            print(i, j, k)
```

А так выглядят два последовательных цикла

```
count = 10
for i in range(count):
    print(i)

for i in range(count):
    print(i)
```

Однако, когда мы перебираем какие-то данные, вместо однобуквенных переменных можно использовать подходящие имена. Например так может выглядеть перебор животных, которые хранятся в массиве данных.

```
animals = ['cat', 'dog', 'wolf', 'rat', 'dragon']
for animal in animals:
    print(animal)
```

Как вы помните читаемость имеет значение.

```
имеем массив животных
для каждого животного в списке животных
    печатаем животное
```

Цикл с нумерацией элементов, функция `enumerate()`

В финале рассмотрим ещё одну функцию, `enumerate()`. Она позволяет добавить порядковый номер к элементам итерируемой последовательности. Доработаем пример с животными. Будем выводить порядковый номер перед указанием животного.

```
animals = ['cat', 'dog', 'wolf', 'rat', 'dragon']
for i, animal in enumerate(animals, start=1):
    print(i, animal)
```

Что изменилось?

- После `for` указано две переменные через запятую. В `i` будет помещаться порядковый номер. В `animal` очередное животное из списка.
- Функция `enumerate()` получила в качестве первого аргумента список животных. Второй аргумент — стартовое значение счётчика, т.е. первое значение, которое попадёт в `i`.

Если второй аргумент не передать, нумерация начнётся с нуля.



Важно! Функция `enumerate` позволяет перебирать только целые числа в порядке возрастания с шагом один.

Задание

Финальное задание с циклами. Перед вами пример программы. Попробуйте мысленно пройти по коду и напишите в чат финальное значение переменной `data`. У вас 2 минуты на каждый вариант кода. Запускать программу не нужно.

1. Вариант кода 1.

```
data = 0
while data < 100:
    data += 2
    if data % 40 == 0:
        break
print(data)
```

2. Вариант кода 2.

```
data = 0
while data < 100:
    data += 3
    if data % 40 == 0:
        break
else:
    data += 5
print(data)
```

3. Вариант кода 3.

```
data = 0
while data < 100:
    data += 3
    if data % 19 == 0:
        continue
    data += 1
    if data % 40 == 0:
        break
else:
    data += 5
print(data)
```

5. Выводы

1. На этой лекции мы разобрали установку и настройку Python. Были изучены правила создания виртуального окружения и работу с pip.
2. Мы повторили основы синтаксиса языка Python. Познакомились с рекомендациями по оформлению кода.
3. Изучили способы создания ветвящихся алгоритмов на Python. Разобрались с разными вариантами реализации циклов.

Краткий анонс следующей лекции

1. Познакомимся со строгой динамической типизацией языка Python.
2. Изучим понятие объекта в Python. Разберёмся с атрибутами и методами объектов.
3. Рассмотрим способы аннотации типов.
4. Изучим "простые" типы данных, такие как числа и строки.
5. Узнаем про математические модули Python.

Домашние задания

1. Установите Python на ваш ПК. Убедитесь, что можете работать как в режиме интерпретатора, так и запускать код в файлах py.
2. Настройте IDE, в которой планируете выполнять задания на семинарах и практические работы. Проверьте возможность запуска программ средствами IDE.

Оглавление

На этой лекции мы	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
1. Простые типы данных и коллекции	4
Функция <code>type()</code>	5
Строгая типизация	5
Функция <code>isinstance()</code>	6
Оператор <code>is</code>	7
Изменяемые и неизменяемые типы и их особенности	7
Задание	9
2. Аннотация типов	10
Модуль <code>typing</code>	11
3. В Python всё объект. Что такое атрибут и метод объекта	12
Методы объекта	13
Функция <code>dir()</code>	13
Функция <code>help()</code>	14
Задание	15
4. Простые объекты	15
Целые числа, функция <code>int()</code>	15
Формат представления числа. Снова о <code>"_"</code> .	16
Функции <code>bin()</code> , <code>oct()</code> , <code>hex()</code>	17
Вещественные числа, функция <code>float()</code>	18
Логические типы, функция <code>bool()</code>	18
Строки, функция <code>str()</code>	19
Способы записи строк	20

Конкатенация строк	21
Размер строки в памяти	22
Методы проверки строк	23
Задание	24
5. Математика в Python	25
Модуль math	25
Модуль decimal	25
Модуль fraction	26
Класс complex()	27
Математические функции "из коробки"	27

На этой лекции мы

1. Познакомимся со строгой динамической типизацией языка Python.
2. Изучим понятие объекта в Python. Разберёмся с атрибутами и методами объектов.
3. Рассмотрим способы аннотации типов.
4. Изучим "простые" типы данных, такие как числа и строки.
5. Узнаем про математические модули Python.

Краткая выжимка, о чём говорилось в предыдущей лекции

1. На прошлой лекции мы разобрали установку и настройку Python. Были изучены правила создания виртуального окружения и работу с pip.
2. Мы повторили основы синтаксиса языка Python. Познакомились с рекомендациями по оформлению кода.
3. Изучили способы создания ветвящихся алгоритмов на Python. Разобрались с разными вариантами реализации циклов.

Термины лекции

- **Коллекция в программировании** — программный объект, содержащий в себе, тем или иным образом, набор значений одного или различных типов, и позволяющий обращаться к этим значениям. Коллекция позволяет записывать в себя значения и извлекать их. Назначение коллекции - служить хранилищем объектов и обеспечивать доступ к ним.
- **Простой тип** — в информатике тип данных, о объектах которого, переменных или постоянных, можно сказать следующее: работа с объектами осуществляется с помощью конструкций языка; внутреннее представление значений объектов может зависеть от реализации транслятора (компилятора или интерпретатора) и от платформы; объекты не включают в себя другие объекты и служат основой для построения других объектов.
- **Хеш** — это криптографическая функция хеширования, которую обычно называют просто хэшем.
- **Хеш-функция** представляет собой математический алгоритм, который может преобразовать произвольный массив данных в строку фиксированной длины, состоящую из цифр и букв.
- **Атрибуты** — это переменные, конкретные характеристики объекта, такие как цвет поля или имя пользователя.
- **Методы** — это функции, которые описаны внутри объекта или класса. Они относятся к определенному объекту и позволяют взаимодействовать с ними или другими частями кода.
- **Дандер (англ. Dunder) или магические методы в Python** — это методы, имеющие два префиксных и суффиксных подчеркивания в имени метода. Дандер здесь означает «Двойное Подчеркивание».
- **ASCII (англ. American standard code for information interchange)** — название таблицы (кодировки, набора), в которой некоторым распространённым печатным и непечатным символам сопоставлены числовые коды.
- **Комплексные числа (от лат. complexus — связь, сочетание)** — числа вида $a+bi$, где a, b — вещественные числа, i — мнимая единица, то есть число, для которого выполняется равенство: $i^2 = -1$.

Подробный текст лекции

1. Простые типы данных и коллекции

В Python существует достаточно большое количество типов данных. И речь идёт не о множестве вариантов целого числа на 1, 2, 4 и 8 байт со знаком и без знака. Целый тип как раз представлен в единственном экземпляре. Условно все типы данных можно разделить на простые типы и коллекции. А вот примитивных типов, вроде того же целого беззнакового на 2 байта и т.п. в Python нет.

Коллекции объединяют в себе другие типы данных. Самый популярный пример коллекции в информатике — массив.

В противоположность коллекциям простые типы не объединяют объекты в группы, а используются сами по себе. О них пойдет речь не этой лекции. Там, где это необходимо, будем вспоминать коллекции. Но подробнее разберём их на следующей лекции.

Строгая динамическая типизация Python

Повторим один момент из первой лекции. Python является языком со строгой динамической типизацией. Это означает что тип объекта изменить невозможно, он строго задаётся при создании объекта. При этом переменные могут ссылаться на объекты разных типов. И это не вызывает ошибки.

```
a = 5
a = "hello world"
a = 42.0 * 3.141592 / 2.71828
```

Функция type()

Чтобы разобраться в динамической типизации, воспользуемся функцией type(). Она возвращает класс объекта, его тип.

```
a = 5
print(type(a))
a = "hello world"
print(type(a))
a = 42.0 * 3.141592 / 2.71828
print(type(a))
```

Как видите у нас меняется класс объекта, на который ссылается переменная. Она динамически изменяется.

Слово class при выводе типа сообщает, что мы обратились к объекту, экземпляру класса. Это не просто пара байт информации, а сложная структура, обеспечивающая работу кода в целом и динамическую типизацию в частности.

Строгая типизация

Вспомним функцию id() из прошлой лекции.


```
a = 5
print(type(a), id(a))
a = "hello world"
print(type(a), id(a))
a = 42.0 * 3.141592 / 2.71828
print(type(a), id(a))
```

Каждый раз возвращается новый адрес объекта в памяти. Python не меняет класс объекта. Он меняет переменную. Каждый раз она ссылается на новый объект. Такая механика прописана в самом интерпретаторе.

Функция `isinstance()`

Для проверки типа объекта используется функция `isinstance()`.

```
isinstance(object, classinfo)
```

Функция принимает на вход объект и класс и возвращает истину, если объект является экземпляром прямого или косвенного подкласса.

```
data = 42
print(isinstance(data, int))
```

Получили `True`, т.к. 42 — целое число.

```
data = True
print(isinstance(data, int))
```

Снова истина, т.к. логический объект `True` в Python подклассом, основанном на классе `int`. Проверка может проходить сразу по нескольким классам. В этом случае истину вернётся, если объект является экземпляром любого из переданных в кортеже классов.

```
data = 3.14
print(isinstance(data, (int, float, complex)))
```

Внимание! Обратите внимание, что функция принимает 2 аргумента. Второй аргумент — кортеж, внутри которого через запятую перечислены 3 класса.



Важно! Для проверки типа можно вызывать функцию `type()` и сравнить значение с результатом. Но документация к языку не рекомендует так делать. Для получения информации о типе объекта правильно использовать функцию `isinstance()`.

Оператор `is`

Ещё один способ взаимодействия с типами данных — оператор `is`. Он похож на сравнение (двойное равно), но сравнивает не значения, а сами объекты.

```
num = 2 + 2 * 2
digit = 36 / 6
print(num == digit)
print(num is digit)
```

По сути оператор `is` занимается сравнением "айдишников" объектов, т.е. проверяет лежит ли объекты в одном месте в памяти компьютера.

Заменяем в примере обычное деление на целочисленное. Python вычислил оба значения на этапе предкомпиляции, понял что это "целое шесть" и создал один объект вместо двух. Теперь оператор `is` возвращает истину, т.к. `num` и `digit` указывают на один и тот же объект в памяти.

Изменяемые и неизменяемые типы и их особенности

Класс объекта зафиксирован. Целое всегда будет целым, строка строкой, а кортеж кортежем. Однако объекты в Python делятся на изменяемые (`mutable`) и неизменяемые (`immutable`). При попытке изменить неизменяемый объект возможны два варианта:

- Замена на новый объект того же типа

```
a = 5
print(a, id(a))
a += 1
print(a, id(a))
```

Переменная `a` ссылается на число 5, которое лежит в определённом месте в ОЗУ. При увеличении значения на единицу был создан новый объект в памяти. Переменная `a` теперь указывает на него.

- Второй вариант развития событий при изменении неизменяемого — вызов ошибки. Строка неизменяема. Попробуем заменить пробел подчеркиванием.

```
txt = 'Hello world!'
txt[5] = '_'
```

Получим `TypeError: 'str' object does not support item assignment` т.к. изменить символ в неизменяемой строке нельзя. Но ведь в Python есть возможность такой замены.

```
txt = 'Hello world!'
print(txt, id(txt))
txt = txt.replace(' ', '_')
print(txt, id(txt))
```

Присмотритесь к адресам. Мы создали новый объект строкового типа, который занимает новое место в памяти. Изменить старый не удалось. К строкам мы вернёмся позже на этой и следующей лекциях.

Запоминаем неизменяемые типы данных

Как понять изменяемый перед нами объект или нет. Вариант 1 — запомнить таблицу.

Неизменяемы	Изменяемые
None	
Числа: int, bool, float, complex	
Последовательности: str, tuple, bytes	Последовательности: list, bytearray
Множества: frozenset	Множества: set
	Отображения: dict

Как вы можете заметить все числа в Python относятся к неизменяемым типам данных. А значит любые математические операции создают новый объект в памяти. Пример ниже служит подтверждением.

```
a = c = 2
b = 3
print(a, id(a), b, id(b), c, id(c))
a = b + c
print(a, id(a), b, id(b), c, id(c))
```

Обратите внимание на первую строку кода. Подобное присваивание допустимо в Python. Переменные `a` и `c` будут указывать на один и тот же объект.

После сложения переменная `a` указывает на новый объект. Переменная `c` по прежнему указывает на число 2, т.к. `int` — неизменяемый тип данных.

Хэш `hash()` как проверка на неизменяемость

Второй вариант убедиться в изменяемости или неизменяемости — проверка на хеширование.

Хеш — это криптографическая функция хеширования, которую обычно называют просто хэшем. **Хеш-функция** представляет собой алгоритм, который может преобразовать произвольный массив данных в набор бит фиксированной длины.

В Python для получения хеша используется функция `hash()`. Если объект является неизменяемым, функция возвращает число — хеш-сумму. Изменяемые объекты хешировать нельзя исходя из самого определения хеш-функции. Если массив данных может меняться, значит будет меняться и хеш-сумма. Подобное поведение нарушало бы логику кода. Рассмотрим на примере.

```
x = 42
y = 'text'
z = 3.1415
print(hash(x), hash(y), hash(z))
my_list = [x, y, z]
print(hash(my_list)) # получим ошибку, т.к. list изменяемый
```

Как видите нижняя строка кода вызывает ошибку `TypeError: unhashable type: 'list'`. Если вдруг забыли изменяемый объект или нет, просто попробуйте получить его хеш.

Задание

Напишите небольшую программу, которая запрашивает у пользователя любой текст и выводит о нём следующую информацию:

- тип объекта,
- адрес объекта в оперативной памяти,
- хеш объекта.

Результат работы пришлите в чат. У вас 5 минут.

2. Аннотация типов

Учитывая динамическую типизацию, Python не требует аннотацию типов переменных перед их использованием. Однако подобная возможность в языке существует. Появилась она относительно недавно. Указание типов полезно программистам, которые перешли из других языков программирования со статической типизации и привыкли объявлять тип переменной. Также аннотация типов упрощает отладку кода. Ведь IDE может подсказать возможные ошибки, если программист присваивает переменной новый, другой тип данных.

Разберём на примере.

```
a: int = 42
b: float = float(input('Введи число: '))
a = a / b
```

После имени переменной ставим двоеточие и через пробел указываем тип данных. Он совпадает с именем функции-класса, но без круглых скобок на конце.

IDE и линтеры выдадут предупреждение для строки 3 Expected type `'int'`, got `'float'` instead. Мы пытаемся в переменную `a`, указанную как хранилище целых чисел сохранить результат деления, т.е. вещественное число. Исправить можно например изменив строку 1 на следующую: `a: float = 42.0`

Далее на курсе мы будем говорить о функциях. Забежим немного вперёд, чтобы рассмотреть пользу указания типов при создании своих функций. Внимание на пример кода:

```
def my_func(data: list[int, float]) -> float:
    res = sum(data) / len(data)
    return res

print(my_func([2, 5.5, 15, 8.0, 13.74]))
```

Аннотация подсказывает, что в качестве аргумента функция получает список, заполненный целыми или вещественными числами. Результат работы функции - вещественное число.



PEP-8! Код до и после функции отделяется двумя пустыми строками.

Начиная с Python 3.10 возможности указания типов расширились. Например можно указать несколько типов через вертикальную черту. Первый пример из этой главы можно записать так:

```
a: int | float = 42
b: float = float(input('Введи число: '))
a = a / b
```

Мы сообщили, что в переменной `a` будем хранить числа, целые или вещественные.

Модуль typing

Для указания типа переменной можно использовать модуль `typing`. Они содержат как готовые типы данных, так и обобщенные. В таблице представлен доступные на текущий момент типы, разделённые по группам. Назначение большинства из них понятно по названиям. А с некоторыми вы никогда не столкнётесь.

Примитивы супер специаль- ного типа	Абсолютные типы из <code>collections.abc</code>	Структур- ные проверки, протоколы	Коллекция конкретных типов	Другие конкретные типы	Одноразо- вые вещи
Annotated, Any, Callable, ClassVar, Final, ForwardRef, Generic,	AbstractSet, ByteString, Container, ContextManager, Hashable, ItemsView,	Reversible, SupportsAbs, SupportsBytes, SupportsComplex, SupportsFloat	ChainMap, Counter, Deque, Dict, DefaultDict, List, OrderedDict,	BinaryIO, IO, Match, Pattern, TextIO	AnyStr, cast, final, get_args, get_origin, get_type_hints, NewType,

Literal, Optional, Protocol, Tuple, Type, TypeVar, Union	Iterable, Iterator, KeysView, Mapping, MappingView, MutableMapping, MutableSequence, MutableSet, Sequence, Sized, ValuesView, Awaitable, AsyncIterator, AsyncIterable, Coroutine, Collection, AsyncGenerator, , AsyncContextManager	, SupportsIndex, SupportsInt, SupportsRound	Set, FrozenSet, NamedTuple, TypedDict, Generator		no_type_check, no_type_check_decorator, NoReturn, overload, runtime_checkable, Text, TYPE_CHECKING
--	---	--	--	--	--

Ещё раз напомню, что программа будет работать без указания типа. Более того, в процессе исполнения Python игнорирует аннотации. Если переменная получит значение неподходящего типа, ошибку это не вызовет. Указание типов служит для повышения читаемости кода и более быстрой отладки.

Если вы будете работать в команде, которая придерживается аннотации, вы знаете где искать. Далее на курсе мы будем использовать указание типа там, где это уместно. Но не будет делать аннотацию обязательной.

3. В Python всё объект. Что такое атрибут и метод объекта

С объектами в Python мы разобрались. Всё есть объект. У объектов могут быть атрибуты и методы. Обычно о них говорят при изучении ООП. Но даже используя процедурный подход к написанию кода мы пользуемся объектами "из коробки", а следовательно можем обращаться к их атрибутам и методам.

Атрибуты объекта

Атрибуты — это переменные, конкретные характеристики объекта, такие как цвет поля или имя пользователя. По сути атрибут хранит информацию о состоянии объекта. Для обращения к атрибуту объекта в Python нужно после его имени поставить точку и далее указать название атрибута. Пока мы не научились создавать свои классы, рассмотрим атрибут на следующем примере.

```
print("Hello world!".__doc__)  
print(str.__doc__)
```

При запуске программы получим два совершенно одинаковых описания работы строк. Что произошло? Мы обратились к магическому атрибуту строки под названием `__doc__`. Данный атрибут хранит строку документации, описывающую объект.

И снова о подчёркивании "_"

Как вы только что заметили, мы обратились к атрибуту, в названии которого использован символ подчёркивания. А если точнее, то два подчёркивания до имени и два после. Такие переменные называют магическими или дандер. Магическим может быть как атрибут, хранящий какие-то данные, так и метод.



Важно! Магические атрибуты и методы более подробно рассмотрим во второй части курса, когда речь пойдёт о ООП. Сейчас о них стоит думать, как о внутренних механизмах, которые обеспечивают работу Python с объектами и позволяют программисту писать короткий код не задумываясь о внутренней реализации.

Методы объекта

Методы — это функции, которые описаны внутри объекта или класса. Они относятся к определённому объекту и позволяют взаимодействовать с ними или другими частями кода. По сути метод — это функция класса.

В Python для обращения к методу используется точечная нотация, как и с атрибутами. Отличии в том, что после имени метода ставятся круглые скобки. А если метод ожидает получить данные на вход, они указываются как аргументы в скобках. Рассмотрим на примере:

```
print("Hello world!".upper())  
print("Hello world!".count('l'))
```


В первой строке мы вызвали метод строки, который пытается привести её к верхнему регистру. Метод `count()` принимает на вход аргумент — строку для поиска. В качестве ответа возвращается целое число, количество вхождений переданной строки внутри исходной.

Функция `dir()`

Один из способов проверки наличия атрибутов и методов у объекта - передача его функции `dir()` в качестве аргумента. Посмотрим на примере строки, раз уж мы с ней активно работаем в этой главе.

```
print(dir("Hello world!"))
```

Получили огромный список значений. Первая половина списка - дандер методы. Их легко отличить по подчёркиваниям. Обычно их не используют напрямую, если не решают узкоспециализированные задачи или не работают с переопределением методов в ООП. Впрочем, переопределение методов — узкоспециализированная задача.

Вторая часть списка — методы, которые доступны программисту для работы со строками. Тут есть `upper` и `count`, которые мы уже использовали и ещё десяток других. Поговорим о строковых методах позже. А пока уточню, что функция `dir()` полезна, если вы пишете код и забыли название нужного вам метода.



Важно! Некоторые IDE выводят информацию об атрибутах и методах объекта после нажатия точки за именем. IDE неявно вызывает функцию `dir()`.

Функция `help()`

Встроенная функция `help()` выводит подсказку об объекте, который передается в качестве аргумента. Таким объектом может быть что угодно, будь то переменная, функция, класс. Можно сказать, что `help()` - более продвинутая версия `dir()`. Ведь программист получает не только имена методов и атрибутов объекта, но и описание того как они работают.

Попробуем вызвать "помощь" для "привет мира".

```
help("Hello world!")
```

На первый взгляд не получили ничего хорошего, документации нет.

```
No Python documentation found for 'Hello world!'.  
Use help() to get the interactive help utility.  
Use help(str) for help on the str class.
```

Но если внимательно прочитать описание, станет понятно что:

1. Функция `help()` без аргументов запускает интерактивный режим. В нём можно указывать имена зарезервированных слов, встроенных функций, модулей и получать справочную информацию.

```
help()
```

Для выхода из режима справки используйте команду `quit`.

2. Если передать в функцию имя класса, получим подробное описание его работы.

```
help(str)
```

Задание

Запустите интерактивный режим справки и проведите два небольших исследования.

1. Введите команду `keywords`, далее любое интересное вам ключевое слово из списка. Прочитайте описание и напишите в чат пару слов о том, что узнали. У вас 3 минуты.
2. Введите команду `symbols`, далее любой заинтересовавший вас символ из списка. Прочитайте описание и напишите в чат пару слов о том, что узнали. У вас 3 минуты.

4. Простые объекты

Рассмотрим подробнее особенности работы с простыми объектами, с числами. В финале лекции поговорим о строках. Их можно считать массивами, т.е. коллекциями. Но так как строки в Python неизменяемы и хранят только символы, назовём их условно простыми.

Целые числа, функция int()

Целое число имеет тип `int`. Для преобразования строки к числу используется одноименная функция.

`int(x, base=10)`.

Первый аргумент — объект, который мы хотим преобразовать в число. Обычно это другое число и или числовая строка. Вторым аргументом указывает на основание системы счисления. По умолчанию используется десятичная система, её можно не указывать. Для `base` допустимы значения от 2 до 36, т.е. от двоичной до тридцати шестиричной системы счисления. Примеры использования.

```
x = int("42")
y = int(3.1415)
z = int("hello", base=30)
print(x, y, z, sep='\n')
```

Мы преобразовали десятичное число из строки в число, отбросили у вещественного числа дробную часть и преобразовали строковую запись числа в тридцатиричной системе счисления в её десятичный числовой аналог.



PEP-8! При указании значений для ключевых аргументов функции пробелы вокруг знака равенства не ставятся.

У целых в Python есть одна полезная особенность. Объект изменяет свои размеры в зависимости от длины целого числа. Переполнения регистра не происходит. В Python "резиновый int". При этом вы должны понимать, что любой объект хранит в себе "служебную информацию", которая также занимает место в памяти. Воспользуемся функцией `getsizeof()` из модуля `sys`, чтобы посмотреть на затраты памяти под целым числом.

```
import sys

STEP = 2 ** 16
num = 1
for _ in range(30):
    print(sys.getsizeof(num), num)
    num *= STEP
```



PEP-8! После импорта модулей ставится пустая строка.

Для хранения "единицы" в 64-х разрядной версии Python тратится 28(!) байт памяти. Это объект со своей служебной информацией и несколькими байтами под само число.

При этом мы можем хранить огромные числа, превышающие long integer на много порядков без проблем и лишних приёмов программирования. Число Гугол, т.е. 10 в степени 100 займет всего лишь 72 байта.

```
print(sys.getsizeof(10 ** 100))
```

Формат представления числа. Снова о "_".

Ещё одна особенность, которая упрощает чтение больших чисел появилась в Python 3.6. Это символ подчеркивания в качестве разделителя групп цифр. Да, снова он. И не в последний раз.

```
num = 7_901_123_456_789
```

Интерпретатор опускает символ подчеркивания. Они нужны лишь для программиста, читающего код.



Внимание! Кроме того, обратите внимание на цикл из примера кода о "резиновом инт".

```
for _ in range(30):
```

Конструкция цикла for in ожидает, что после for указывается переменная для приёма значений, которые берутся из итератора указанного после in. Но если внутри цикла значения не нужны, в качестве имени переменной используют подчеркивание "_". Важно понимать, что использование подчеркивания в теле цикла неверно. Скорее всего вам нужна переменная i, item или другая подобная. Если подчёркиваем, то только один раз.

Функции bin(), oct(), hex()

Помимо десятичной системы у Python есть функции для строкового представления чисел в двоичной, восьмеричной и шестнадцатеричной системах счисления.

```
num = 2 ** 16 - 1
b = bin(num)
o = oct(num)
```

```
h = hex(num)
print(b, o, h)
```

Обратите внимание на приставку в начале числа, состоящую из нуля и латинской строчной буквы: 0b — двоичное, 0o — восьмеричное, 0x — шестнадцатеричное представление. Подобная запись сразу говорит о системе счисления. А само представление в формате, отличном от десятичного, применяется для решения узкого спектра задач.



PEP-8! В качестве имени переменной мы использовали латинскую строчную "o". Это допустимое имя. Но никогда не используйте символы 'l' (строчная буква "л"), 'O' (заглавная буква "О") или 'I' (заглавная буква "Ай") в качестве имен переменных, состоящих из одного символа. В некоторых шрифтах эти символы неотличимы от цифр один и ноль.

Вещественные числа, функция float()

Числа с плавающей запятой представлены классом float. Для хранения таких чисел ПК, а не только Python используют особый формат представления числа: мантисса и порядок. Так число 23321.345 правильнее было бы представить как $2.3321345 \cdot 10^{-4}$.

Особенности подобного формата хранения чисел могут приводить к погрешностям вычисления.

```
print(0.1 + 0.2)
```

Вместо ожидаемого 0.3 получили 0.30000000000000004. При округлении это будет тот же 0.3. Но когда пишешь код, ожидаешь получить верный результат сразу.

Вторая особенность вещественных чисел - ограничение на хранения информации. Попробуем сохранить достаточно большое по количеству знаков число с плавающей точкой.

```
pi = 3.141592653589793238462643383279502884197169399375
print(pi)
```

При выводе на печать, а если быть точным, то в момент сохранения объекта в память в качестве вещественного числа произошло отбрасывание части цифр. В

итоге в памяти осталось 3.141592653589793. И, да, вы верно заметили. Подчёркивание можно использовать для удобства чтения чисел любого типа.

В случае с математикой вещественных чисел в Python стоит ожидать подобных погрешностей вычисления и округления. О том как их избежать узнает в конце сегодняшней лекции. Пока же стоит понять, что float является компактным и быстрым типом данных для математических операций с плавающей запятой, если нам не требуется высокая точность результата.

Логические типы, функция bool()

С True и False мы уже знакомы. Это две неизменные константы, которые являются ответвлением числового типа. Функция bool() возвращает одно из двух значений, в зависимости от входного значения. У "логики" есть несколько полезных особенностей, которые облегчают разработку.

Все числа преобразуются к истине, за исключением нуля. Он считается ложью.

```
DEFAULT = 42
num = int(input('Введите уровень или ноль для значения по
умолчанию: '))
level = num or DEFAULT
print(level)
```

Любая строка текста также приводится к истине. Ложью является пустая строка (не путайте с пробелом), т.е. строка длиной в ноль символов.

```
name = input('Как вас зовут? ')
if name:
    print('Привет, ' + name)
else:
    print('Анонимус, приветствую')
```

Коллекции, о которых будем подробно говорить на следующей лекции приводятся к истине, если в них есть элементы. Пустая коллекция — ложь.

```
data = [0, 1, 1, 2, 3, 5, 8, 13, 21]
while data:
    print(data.pop())
```

Работу данного примера подробнее разберём на следующем уроке. Но если коротко, в цикле мы удаляем элементы списка до тех пор пока список не окажется пустым.



Внимание! Обратите внимание, что во всех примерах происходило неявное приведение к логическому типу, мы явно не использовали функцию `bool()`.

Строки, функция `str()`

В Python нет типа данных символ. Есть только класс `str`. В нём можно хранить как один символ, аналог `char` в си-подобных языках, так и любое большее количество символов. Почему символы, а не буквы? Для хранения используется кодировка `utf-8`, т.е. в строке могут быть не только буквы, цифры, знаки препинания, но и иероглифы, смайлики и всё то, что хранится в кодировке Unicode.

При работе со строками стоит помнить, что это неизменяемый тип данных. При этом `str` можно представить как коллекцию символов — массив. Выходит, что строка является коллекцией? Верно.

На этой лекции рассмотрим несколько приёмов работы со строками как с неизменяемыми объектами. А на следующей поговорим о строке как о коллекции символов.

Способы записи строк

Python допускает одинарные или двойные кавычки для записи строки. Одни кавычки могут включать другие, но они не должны смешиваться.

```
txt = 'Книга называется "Война и мир".'
```

Тройные двойные кавычки позволяют писать текст в несколько строк. Если такой текст не присвоить переменной, он превращается в многострочный комментарий. Подобные комментарии используют для создания документации к коду. Например так:

```
class str(object):  
    """
```

```
str(object='') -> str
str(bytes_or_buffer[, encoding[, errors]]) -> str

...
"""
```

Ещё один способ записать строки, перечислить их последовательно друг за другом.

```
text = 'Привет.' 'Как ты, друг?' 'Рад тебя видеть.'
print(text)
```

Такой приём работает, но считается плохим стилем. Плюс, а точнее минус, строки соединились без пробелов.

На этом способы записи строк не заканчиваются. Посмотрите на этот код:

```
very_long_text = 'Lorem ipsum dolor sit amet, consectetur
adipisicing elit. A ab alias animi assumenda at aut ' \
                 'commodi, consequatur cumque ea harum, hic id
illum ipsam itaque laboriosam magnam minus nam nulla ' \
                 'numquam obcaecati officia officiis porro
possimus praesentium quaerat temporibus ullam veniam? '
```

Как вы помните, PEP8 рекомендует не писать более 120 символов в одной строке. Обратный слеш “\” позволяет писать продолжение с новой строки. Требования в 120 символов выполняются. А Python воспринимает несколько строк как одну.



Важно! Подобный приём с обратным слешем работает не только для строк текста, но и для кода. Очень длинную строку можно разделить записать в несколько. Но помните, что зачастую такие строки трудно читать. Возможно не стоит лепить решение в одну мега строчку, а разбить его на более читаемый вариант.

Конкатенация строк

Отдельный способ создания строк — конкатенация. Или говоря проще, сложение. Разберём на примере и обсудим все его нюансы:

```
LIMIT = 120
ATTENTION = 'Внимание!'
```



```

name = input('Твоё имя? ')
age = int(input('Твой возраст? '))
text = ATTENTION + ' Хотя тебе и осталось ' + str(100 - age) + \
      " до ста лет, но длина строки не должна превышать " +
str(LIMIT) + ' символов.'
print(text)

```

Переменная `text` получилась в результате сложения нескольких строк. При этом:

- Все элементы должны быть строками. Иначе получим ошибку вида: `TypeError: can only concatenate str (not "int") to str` Именно для этого мы обернули переменные с числами, такие как `LIMIT`, функцией `str()`
- Если в переменной хранится строковое значение, оборачивать её в `str()` не нужно. Получится масло масляное.
- Складывать можно строки в одинарных, двойных и даже в трех двойных кавычках. Но лучше выбрать единый стиль записи строк и придерживаться его во всём проекте.
- Если код не влезит в 120 символов строки, не стесняемся использовать обратный слеш для его разделения на несколько строк.

Конкатенация строк затратна по памяти и по времени. Для простых задач допустим подобный подход. В реальных проектах конкатенация используется для формирования констант. В остальных случаях используют способы форматирования строк. О них поговорим на следующей лекции.

Кстати, рекомендации Google по стилю кода, которые являются продвинутой версией PEP-8 с дополнительными требованиями запрещают программистам использовать конкатенацию строк, особенно если она происходит внутри цикла. Почему? Поговорим о размере строки в памяти.

Размер строки в памяти

Строки как объекты тратят память на служебную информацию, а как массивы на хранение текста. В 64-х разрядной версии Python служебная информация занимает 48 байт. Разберём пример кода:

```

empty_str = ''
en_str = 'Text'
ru_str = 'Текст'
unicode_str = '😄😍😏😞'
print(empty_str.__sizeof__())
print(en_str.__sizeof__())
print(ru_str.__sizeof__())
print(unicode_str.__sizeof__())

```

Во-первых обратите внимание на магический метод `__sizeof__()`. Он работает аналогично `sys.getsizeof` и возвращает количество байт занятых объектом. Почему же пустая строка заняла 49 байт, если служебная информация использует 48? Один байт - символ конца строки.

Теперь посмотрим на текст. Английские буквы тратят по одному байту на символ. Если же речь идёт о русском языке или любых других символах, кодирование занимает 2 или 4 байта. Это особенность хранения информации в кодировке UTF-8 и фишка языка Python для доступа к букве по индексу.

А теперь зная, что строка тратит много памяти, что строка неизменяемый тип данных и что при конкатенации строк создаются новые объекты, которые занимают дополнительную память вы сами можете сделать вывод почему сложение строк не приветствуется.

Методы проверки строк

Ряд методов анализируют строку текста и возвращают истину или ложь в зависимости от содержимого строки. Часто используемые методы перечислены в таблице:

Название метода	Описание
<code>str.isalnum()</code>	Возвращает True, если все символы в строке буквенно-цифровые. Символ является буквенно-цифровым, если одно из следующих значений возвращает True: <code>c.isalpha()</code> , <code>c.isdecimal()</code> , <code>c.isdigit()</code> или <code>c.isnumeric()</code> .
<code>str.isalpha()</code>	Возвращает True, если все символы в строке являются буквенными. Алфавитные символы — это символы, определенные в базе данных символов Юникода как «буква»
<code>str.isdecimal()</code>	Возвращает True, если все символы в строке являются десятичными символами
<code>str.isdigit()</code>	Возвращает True, если все символы в строке являются цифрами. Цифры включают десятичные символы и цифры, требующие специальной обработки, например цифры надстрочного индекса совместимости.
<code>str.isnumeric()</code>	Возвращает True, если все символы в строке являются

Название метода	Описание
str.isalnum()	Возвращает True, если все символы в строке буквенно-цифровые. Символ является буквенно-цифровым, если одно из следующих значений возвращает True: c.isalpha(), c.isdecimal(), c.isdigit()или c.isnumeric().
	числовыми символами. Числовые символы включают цифровые символы и все символы, которые имеют свойство числового значения Unicode
str.isascii()	Возвращает True, если строка пуста или все символы в строке ASCII
str.islower()	Возвращает True, если все символы в строке в нижнем регистре
str.istitle()	Возвращает True, если строка является строкой с заглавным регистром и содержит хотя бы один символ
str.isupper()	Возвращает True, если все символы в строке в верхнем регистре
str.isprintable()	Возвращает True, если все символы в строке доступны для печати или строка пуста. Непечатаемые символы — это символы, определенные в базе данных символов Unicode как «Другие» или «Разделители», за исключением пробела ASCII (0x20), который считается печатаемым.
str.isspace()	Возвращает True, если в строке есть только пробельные символы



Внимание! Обратите внимание, что методы начинаются с приставки is. Подобный приём часто используется в программировании. Если переменная содержит логическое значение или функция/метод возвращает логическое значение, в начале добавляют приставку is — намёк на истину или ложь в качестве результата.

Задание

Напишите небольшую программу, которая запрашивает у пользователя текст. Если текст можно привести к целому числу, выведите его двоичное, восьмиричное и шестнадцатеричное представление. А если преобразование к целому невозможно, сообщите написан ли текст в ASCII или нет. Результат работы отправьте в чат. У вас 7 минут.

5. Математика в Python

Немного математики в финале лекции. Не пугайтесь, если что-то подзабыли. Разберём в общих чертах математические возможности Python. В решение сложных математических задач углубляться не будем.

В Python есть несколько модулей в стандартной библиотеке, которые облегчают математические расчёты. Для доступа к ним необходимо выполнить импорт в начале файла.

```
import math
import decimal
import fractions
```

Модуль math

В математическом модуле содержатся константы, например число "пи", "е", бесконечность и др.

```
print(math.pi, math.e, math.inf, math.nan, math.tau, sep='\n')
```

Кроме того внутри модуля есть множество математических функций: синусы, косинусы, тангенсы, логарифмы, факториал.

Вы можете подробно изучить многообразие модуля самостоятельно. В этом вам помогут функции `dir()` и `help()`

Модуль decimal

Помните пример про точность вещественных чисел: `print(0.1 + 0.2)` ?

Модуль `decimal` позволяет хранить числа с плавающей запятой и проводить с ними математические вычисления без потери точности и ошибок преобразования. Для этого надо воспользоваться классом `Decimal` из модуля. Рассмотрим на примере.

```
pi =
decimal.Decimal('3.141_592_653_589_793_238_462_643_383_279_502_88
4_197_169_399_375')
print(pi)
num = decimal.Decimal(1) / decimal.Decimal(3)
print(num)
```

Передав в качестве аргумента число π в виде строки, мы получаем его математическое представление без потери точности. Помимо строки аргументом может быть целое или вещественное число. Например разделив 1 на 3 мы получаем вещественно число с большой точность знаков после запятой.

Часто при работе с модулем указывают точность вычислений. По умолчанию она равна 28 знакам до и после запятой. Например нам надо что-то посчитать с точность 120 знаков:

```
decimal.getcontext().prec = 120
science = 2 * pi * decimal.Decimal(23.452346) ** 2
print(science)
```

В первой строке кода задали точность через обращение к атрибуту `prec` у метода `getcontext()`. Дальше считаем как обычно.



Важно! Если вы планируете писать код для банковской системы или любой другой, где важна точность расчётов, используйте модуль `decimal` вместо класса `float`.

Модуль `fraction`

Для работы с дробями есть модуль `fractions`. Откроем пример, чтобы сразу стало ясно как он работает.

```
f1 = fractions.Fraction(1, 3)
print(f1)
f2 = fractions.Fraction(3, 5)
print(f2)
print(f1 * f2)
```

Передаём в класс `Fractions` через запятую числитель и знаменатель дроби. Далее работаем как с обычными числами. Модуль полезен при работе с бесконечными дробями, когда `decimal` не обеспечивает необходимую точность хранения числа.

Класс `complex()`

Если вы проходили комплексные числа в школе или вузе, поздравляю. Python позволяет работать и с ними. Если нет — не переживайте. Скажу о них всего пару слов. Комплексные числа — числа с мнимой единицей, квадратным корнем из минус одного.

```
a = complex(2, 3)
b = complex('2+3j')
print(a, b, a == b, sep='\n')
```

В каждом случае получим `(2+3j)` — комплексное число в Python.



Внимание! Обратите внимание на то, что Python для обозначения мнимой единицы использует букву "j", а в математике принято использовать "i". Во всём остальном математические операции в Python совпадают с классической математикой.

Математические функции "из коробки"

В финале лекции несколько слов о математических функциях "из коробки".

- `abs(x)` — возвращает абсолютное значение числа `x`, число по модулю.
- `divmod(a, b)` — функция принимает два числа в качестве аргументов и возвращает пару чисел — частное и остаток от целочисленного деления. Аналогично вычислению `a // b` и `a % b`.
- `pow(base, exp[, mod])` — при передаче 2-х аргументов возводит `base` в степень `exp`. При передаче 3-х аргументов, результат возведения в степень делится по модулю на значение `mod`.
- `round(number[, ndigits])` — округляет число `number` до `ndigits` цифр после запятой. Если второй аргумент не передать, округляет до ближайшего целого.

6. Выводы

На этой лекции мы:

1. Познакомились со строгой динамической типизацией языка Python.
2. Изучили понятие объекта в Python. Разобрались с атрибутами и методами объектов.
3. Рассмотрели способы аннотации типов.
4. Изучили "простые" типы данных, такие как числа и строки.
5. Узнали про математические возможности Python.

Краткий анонс следующей лекции

1. Разберёмся что такое коллекция и какие коллекции есть в Python
2. Изучим работу со списками, как с самой популярной коллекцией.
3. Узнаем как работать со строкой в ключе коллекции
4. Разберём работу с кортежами
5. Узнаем что такое словари и как с ними работать
6. Изучим множества и особенности работы с ними
7. Разберём работу с байтами как с массивами

Домашнее задание

Поработайте со справочной информацией в Python, функцией `help()`. Попробуйте найти зарезервированные слова, функции и модули, которые прошли за две лекции и почитать про них. Если вы плохо читаете на английском, воспользуйтесь любым онлайн переводчиком.

Оглавление

На этой лекции мы	4
Дополнительные материалы к лекции	4
Краткая выжимка, о чём говорилось в предыдущей лекции	4
Термины лекции	5
1. Списки, list	5
Доступ к элементам списка	7
Метод extend	9
Метод pop	9
Метод count	10
Метод index	11
Метод insert	11
Метод remove	12
Сортировка списков	12
Функция sorted()	13
Метод sort()	13
Разворот списков	14
Функция reversed()	14
Метод reverse() и синтаксический сахар[::-1]	14
Срезы	15
Метод copy()	16
Зачем нужна функция copy.deepcopy()	16
Плюсы и минусы создания копии	17
Функция len	18
Задание	18
2. Строки, str	19
Работа со строками как с массивами	19

Методы count, index, find	19
Реверс строк	20
Форматирование строк	20
Метод format	21
Уточнение формата	22
Методы строк	23
Метод split	23
Метод join	24
Методы upper, lower, title, capitalize	24
Методы startswith и endswith	24
Задание	25
3. Кортеж, tuple	25
Способы создания кортежа	25
Кортежи реализуют все общие операции последовательностей	26
Задание	26
4. Словарь, dict	27
Способы создания словаря	27
Добавление нового ключа	28
Доступ к значению словаря	
Доступ через квадратные скобки []	28
Доступ через метод get	29
Часто используемые методы словарей	29
Метод setdefault	29
Метод keys	30
Метод values	30
Метод items	31
Метод popitem	31
Метод pop	32
Метод update	32
Задание	33

5. Множества set и frozenset	33
Методы множеств	34
Метод add	34
Метод remove	34
Метод discard	35
Метод intersection	35
Метод union	36
Метод difference	36
Проверка на вхождение, in	36
Задание	37
6. Классы bytes и bytearray	37
7. Вывод	38

На этой лекции мы

1. Разберёмся, что такое коллекция и какие коллекции есть в Python.
2. Изучим работу со списками, как с самой популярной коллекцией.
3. Узнаем, как работать со строкой в ключе коллекции.
4. Разберём работу с кортежами.
5. Узнаем, что такое словари и как с ними работать.
6. Изучим множества и особенности работы с ними.
7. Познакомимся с классами байт и массив байт.

Дополнительные материалы к лекции

Форматирование строк

<https://docs.python.org/3/library/string.html#format-specification-mini-language>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Познакомились со строгой динамической типизацией языка Python.
2. Изучили понятие объекта в Python.
Разобрались с атрибутами и методами объектов.
3. Рассмотрели способы аннотации типов.
4. Изучили "простые" типы данных, такие как числа и строки.
5. Узнали про математические возможности Python.

Термины лекции

- **Коллекция** — это программный объект который объединяет в себе несколько более простых объектов
- **LIFO** (англ. last in, first out, «последним пришёл — первым ушёл») — способ организации и манипулирования данными относительно времени и приоритетов.
- **Массив** — упорядоченный набор элементов, каждый из которых хранит одно значение, идентифицируемое с помощью одного или нескольких индексов.
- **Список в информатике** — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза.

Подробный текст лекции

Введение

Сегодня на лекции мы поговорим о коллекциях. Напомню, что коллекция — это программный объект, который объединяет в себе несколько более простых объектов. В Python одна коллекция может являться частью другой коллекции. При этом программист может работать с коллекцией как с единым объектом, обращаться к элементам коллекции и пользоваться встроенными в коллекцию методами и атрибутами.

1. Списки, list

List, список является самой часто используемой коллекцией в Python. Прежде чем говорить о списках, я напомним, что такое массив в информатике. Массив - это непрерывная область в оперативной памяти компьютера, поделённая на ячейки одинакового размера хранящие данные одного типа. Массивы могут быть статическими, то есть размер массива нельзя изменить, и динамическими, когда размер массива изменяется при добавлении или удалении элементов. Один из самых больших плюсов в работе с массивами — это доступ к любой из его ячеек за константное время.

Массив — упорядоченный набор элементов, каждый из которых хранит одно значение, идентифицируемое с помощью одного или нескольких индексов. В простейшем случае массив имеет постоянную длину и хранит единицы данных одного и того же типа, а в качестве индексов выступают целые числа.

В информатике, **список (англ. list)** — это абстрактный тип данных, представляющий собой упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза. Экземпляр списка является компьютерной реализацией математического понятия конечной последовательности. Экземпляры значений, находящихся в списке, называются элементами списка (англ. item, entry либо element); если значение встречается несколько раз, каждое вхождение считается отдельным элементом.

Почему list "список", а не "массив"

Если вы уже пользовались питоновскими списками, то могли задуматься о путанице в понятиях. Ведь список по описанию очень похож на динамический массив.

Пройдемся по основным характеристикам и сравним его с массивом и со списком:

- Единый блок в памяти — массив
- Хранит значения любых типов вперемешку — не массив???
- Добавление элементов в конец — динамический массив
- Доступ к элементам по индексу с константной сложностью — массив

Стоп! Доступ к элементам по индексу за $O(1)$ возможен, если каждая ячейка имеет одинаковый размер. Но как в таком случае хранить разные элементы — делать ячейку по максимальному из хранимых элементов? В таком случае будет много

пустоты. А если добавится новый элемент и он больше всех уже хранящихся? Проводить переупаковку?

Правда в том, что массив хранит значения одного типа — указатели. И все указатели занимают фиксированный размер в 1, 2 (в далёких 90-х), 4 или 8 байт. Зависит от ОС и её разрядности, а также от версии Python и его разрядности.

Указатель — это переменная, в которой хранится адрес объекта в памяти.

И в этом месте мы понимаем, где список работает как список. Обращаясь к элементу массива по индексу мы получаем указатель на реальный объект, хранящийся отдельно. Благодаря этому приему список может смешивать объекты разных типов и размеров внутри себя.

Работа со списками

Прежде чем разберём основные методы списков, поговорим о способах его создания.

Основной — вызов функции `list()`. Без аргументов функция возвращает пустой список. Если передать последовательность, функция вернёт список содержащий переданные элементы.

Однако есть более простой и привычный для большинства программистов способ создания списков. Используются квадратные скобки, внутри которых перечисляются элементы списка. Квадратные скобки — синтаксический сахар, который не только короче пишется, но и работает быстрее.

<https://habr.com/ru/post/671602/>

```
list_1 = list()
list_2 = list((3.14, True, "Hello world!"))
list_3 = []
list_4 = [3.14, True, "Hello world!"]
```

Кроме того список можно создать из другой последовательности. Достаточно передать её функции `list`.

```
new_list = list(other_iterator)
```



Важно! Преобразование одной коллекции в другую имеет линейную асимптотику, т.е. $O(n)$. И по времени, и по памяти конечно же. Ведь мы создаём объект списка и копируем в него данные.

Доступ к элементам списка

Как мы уже разобрались список ведёт себя как динамический массив. А это значит, что к любому элементу списка можно обратиться по индексу. В Python, как и в большинстве языков программирования, индексация начинается с нуля, то есть, первый элемент лежит в ячейке под индексом 0, следующий элемент в ячейке один и так далее.

```
my_list = [2, 4, 6, 8, 10, 12]
print(my_list[0])
print(my_list[6])  # IndexError: list index out of range
```

При попытке обратиться к несуществующему элементу получим ошибку `IndexError`. Положительный индекс считается слева направо. Если индекс отрицательный вычисление элемента идёт справа налево то есть от конца к началу.



Важно! Вы же знаете что в математике нет деления на ноль и минус ноль. Самый правый элемент списка, т.е. последний элемент, имеет индекс - 1, Предпоследний - -2 и т.д.

```
my_list = [2, 4, 6, 8, 10, 12]
print(my_list[-1])
print(my_list[-10])  # IndexError: list index out of range
```

Как и в случае с положительными индексами, попытка обратиться к элементу за пределами списка через отрицательный индекс вызывает ошибку.

Метод `append`

Для добавления нового элемента в конец списка используется метод `append`. Метод принимает один аргумент — объект который будет добавлен в конец динамически увеличенного списка.

```
a = 42
b = 'Hello world!'
c = [1, 3, 5, 7]
my_list = [None]
my_list.append(a)
print(my_list)
my_list.append(b)
```

```
print(my_list)
my_list.append(c)
print(my_list)
```

Обратите внимание, что при добавлении списка `c` в список `my_list` он оказался целиком в одной ячейке списка. Ссылочная система Python позволяет формировать структуры любой глубины вложенности.

Ниже пример плохого кода. Мы добавили в список сам себя.

```
my_list.append(my_list)
print(my_list)
# Вывод: [None, 42, 'Hello world!', [1, 3, 5, 7], [...]]
```

Многоточие в выводе говорит о рекурсивной ссылке. Явный намёк на то, что программист сделал что-то неверно своей программе.

Метод `append` имеет константную асимптотику $O(1)$.

Метод `extend`

Метод `extend` ведёт себя аналогично `append`, то есть добавляет элементы в конец списка. В качестве аргумента `extend` принимает последовательность, итерируется по ней слева направо и каждый элемент добавляет в новую ячейку списка.

```
a = 42
b = 'Hello world!'
c = [1, 3, 5, 7]
my_list = [None]
my_list.extend(a)  # TypeError: 'int' object is not iterable
print(my_list)
my_list.extend(b)
print(my_list)
my_list.extend(c)
print(my_list)
my_list.extend(my_list)
print(my_list)
```

- **`extend(a)`** — если в метод передать не коллекцию, получим ошибку `TypeError`.
- **`extend(b)`** — строка воспринимается как коллекция, в результате каждый символ строки помещается в новую ячейку списка.

- **extend(c)** — итерируемся по списку, с последовательно добавляя его элементы в список `my_list`
- **extend(my_list)** — удваиваем список, добавляя копию всех его элементов.

Метод `extend` имеет константную асимптотику $O(1)$ на добавление одного элемента и зависит от количества переданных объектов, т.е. $O(n)$, где n - количество элементов в последовательности.

Метод `pop`

Метод `pop` позволяет удалить элемент списка. Удаляемый элемент возвращается как результат работы метода.

```
my_list = [2, 4, 6, 8, 10, 12]
spam = my_list.pop()
print(spam, my_list)
eggs = my_list.pop(1)
print(eggs, my_list)
err = my_list.pop(10)    # IndexError: pop index out of range
```

Метод `pop` может принимать на вход индекс удаляемого элемента. Этот элемент вернётся как результат работы метода, т.е. может быть сохранён в переменную. Индекс может быть отрицательным. Тогда элемент считается от правой части списка, с конца.

Если указать индекс выходящий за границы списка получим ошибку `IndexError`. Метод `pop` без аргумента имеет константную асимптотику $O(1)$. Метод `pop` с аргументом имеет линейную асимптотику. Список не допускает пустых ячеек, поэтому при удалении элемента из середины списка все элементы находящиеся правее сдвигаются на одну ячейку влево. Получаем $O(n)$, где n - количество элементов правее удаляемого.

Python Style! Переменные `spam` и `eggs` используется в языке Python как временные переменные. Эти названия, ветчина и яйца, используют True Python разработчики. Ведь язык назван в честь комедийного шоу Летающий цирк Монти Пайтон. А одно из их видео посвящено шуткам про ветчину и яйца.

Метод count

Метод count подсчитывает вхождение элемента в список.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 2, 4, 14, 2]
spam = my_list.count(2)
print(spam)
eggs = my_list.count(3)
print(eggs)
```

Метод принимает именно объект, а не индекс. Если объект отсутствует в списке, count возвращает ноль — элемент был встречен в списке ноль раз.

Count имеет линейную асимптотику $O(n)$, т.к. для подсчёта метод перебирает все элемента списка и сравнивает их с переданным объектом.

Метод index

Метод index возвращает индекс переданного объекта внутри списка.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 2, 4, 14, 2]
spam = my_list.index(4)
print(spam)
eggs = my_list.index(4, spam + 1, 90)
print(eggs)
err = my_list.index(3)  # ValueError: 3 is not in list
```

Метод работает до первого вхождения, т.е. если элемент встречается несколько раз, возвращается индекс первой встречи. Метод может принимать два дополнительных аргумента: индекс start и индекс stop. В таком случае поиск элемента осуществляется внутри заданного диапазона. Если элемент отсутствует в списке, метод вызывает ошибку ValueError.

Чем ближе элемент к началу списка, тем быстрее работает метод. В лучшем случае получаем константную асимптотику $O(1)$, в худшем линейную $O(n)$.



Внимание! Если правая граница, stop индекс, окажется больше, чем количество элементов в списке, поиск будет осуществляться до конца списка. Ошибку это не вызовет.

Метод insert

Метод insert принимает на вход два аргумента — индекс для вставки и объект вставки. Метод добавляет элемент после индекса.

```
my_list = [2, 4, 6, 8, 10, 12]
my_list.insert(2, 555)
print(my_list)
my_list.insert(-2, 13)
print(my_list)
my_list.insert(42, 73)  # my_list.append(73)
print(my_list)
```

Если индекс положительный, элемент добавляется в указанную ячейку, а все последующие элементы списка сдвигаются на ячейку правее.

Если индекс отрицательный, отсчитывается необходимое количество элементов справа для вставки. Например -2 означает, что после вставки справа от добавленного элемента будет находиться ещё два.

В том случае, когда индекс оказывается больше, чем количество элементов списка, объект добавляется в конец. В таком случае логичнее использовать метод append, выполняющий добавление элемента в конец списка.

Метод remove

Метод remove принимает на вход объект, производит его поиск в списке и удаляет в случае нахождения.

```
my_list = [2, 4, 6, 8, 10, 12, 6]
my_list.remove(6)
print(my_list)
my_list.remove(3)  # ValueError: list.remove(x): x not in list
print(my_list)
```

Если удаляемый элемент встречается в списке несколько раз, удаляется только один элемент — самый левый.

А если удаляемый элемент отсутствует в списке, будет вызвана ошибка ValueError.

Сортировка списков

Одна из частых операций при работе со списками их сортировка. Python позволяет отсортировать список на месте, inplace, т.е. не создавая копию. А можно создать копию отсортированного списка как отдельный объект.



Важно! При сортировке элементы списка должны быть одного типа. Иначе Python может не понять как сравнивать между собой элементы разных типов и вызовет ошибку.

```
my_list = ['H', 'e', 'l', 'l', 'o', 1, 3, 5, 7]
my_list.sort() # TypeError: '<' not supported between instances of 'int' and 'str'
res = sorted(my_list) # TypeError: '<' not supported between instances of 'int' and 'str'
```

Функция sorted()

Функция sorted принимает на вход любую коллекцию по которой можно итерироваться и возвращает отсортированный список.



Важно! Функция sorted может принимать не только списки, но и другие последовательности: строки, множества, кортежи, словари и т.п.. При этом функция всегда возвращает список.

```
my_list = [4, 8, 2, 9, 1, 7, 2]
sort_list = sorted(my_list)
print(my_list, sort_list, sep='\n')
rev_list = sorted(my_list, reverse=True)
print(my_list, rev_list, sep='\n')
```

Переданная в функцию коллекция остаётся неизменной после результата работы функции. Если в функцию передать дополнительный аргумент reverse=True, сортировка происходит по убыванию.

Внутри функции используется алгоритм сортировки Timsort — гибридная устойчивая сортировка с временной асимптотикой $O(n \log n)$. Дополнительно тратится $O(n)$ памяти на создание нового отсортированного списка.

Метод sort()

Метод sort осуществляет сортировку элементов списка без создания копии, inplace.

```
my_list = [4, 8, 2, 9, 1, 7, 2]
my_list.sort()
print(my_list)
my_list.sort(reverse=True)
print(my_list)
```

Как и функция sorted метод sort упорядочивает элементы по возрастанию. Если передать дополнительный параметр reverse=True, будет произведена сортировка по убыванию. Внутри метода работает тот же самый алгоритм сортировки Timsort. Но память на создание копии списка мы не тратим.

Разворот списков

Python поддерживает операции по развороту списка. Первый элемент становится последним, второй — предпоследним и так далее.

Функция reversed()

Функция принимает на вход последовательность, которая поддерживает порядок элементов, возвращает функция объект итератор с обратным порядком элементов.

```
my_list = [4, 8, 2, 9, 1, 7, 2]
res = reversed(my_list)
print(type(res), res)
rev_list = list(reversed(my_list))
print(rev_list)
```

Получается, что результат работы функции напрямую не использовать? Если нам нужен новый развёрнутый список, объект итератор стоит обернуть в функцию list. В таком случае мы получим новый развёрнутый список.



Важно! Подобный приём затратен по времени и по памяти.

Обычно функция `reversed` используется в сочетании с циклом `for in`. Такой приём позволяет работать с элементами списка внутри цикла в обратном порядке.

```
for item in reversed(my_list):  
    print(item)
```

Метод `reverse()` и синтаксический сахар `[::-1]`

Если нам нужно развёрнутая версия списка логичнее и удобнее использовать встроенный метод `reverse`.

```
my_list = [4, 8, 2, 9, 1, 7, 2]  
my_list.reverse()  
print(my_list)
```

Метод разворачивает список на месте не создавая копии.

Кроме этого в Python есть возможность получить развёрнутую копию через особую запись в квадратных скобках, синтаксический сахар. После имени списка в квадратных скобках слитно записываем два двоеточия и минус один.

```
my_list = [4, 8, 2, 9, 1, 7, 2]  
new_list = my_list[::-1]  
print(my_list, new_list, sep='\n')
```

Подобная запись аналогична `rev_list = list(reversed(my_list))` рассмотренной выше. Разворот списка с использованием квадратных скобок — частный случай срезов.

Создание копий

Мы уже разобрали способы создания отсортированной копии списка, развёрнутой копии списка. На этом возможности копирования не заканчиваются.

Срезы

Используя квадратные скобки можно делать частичные копии списка - срезы.

Базовый синтаксис следующий.

`list[start:stop:step]`

`start` указывает на первый индекс, который включается в срез. При отсутствии значения `start` равен нулю, началу списка.

`stop` указывает на последний индекс, который не включается в срез. При отсутствии значения `stop` равен последнему элементу списка и включает его в срез.

`step` — шаг движения от `start` до `stop`. По умолчанию `step` равен единице, все элементы по порядку.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
print(my_list[2:7:2])
print(my_list[:7:2])
print(my_list[2::2])
print(my_list[2:7:])
print(my_list[8:3:-1])
print(my_list[3::])
print(my_list[:7:])
```

Метод `copy()`

Метод `copy` создаёт поверхностную копию списка. Начнём с плохого примера, чтобы понять пользу копий.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
new_list = my_list
print(my_list, new_list, sep='\n')
my_list[2] = 555
print(my_list, new_list, sep='\n')
```

Мы скопировали в переменную `new_list` указатель на список `my_list`. Далее мы изменили элемент в исходном списке. Новый список также оказался изменённым. Как вы помните `list` — изменяемый тип данных и подобное поведение нормально. Что делать, если нужно менять оригинал, но не затрагивать копию. Верно. Метод `copy`.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
new_list = my_list.copy()
print(my_list, new_list, sep='\n')
my_list[2] = 555
print(my_list, new_list, sep='\n')
```

Теперь изменяется лишь один список.

Зачем нужна функция `copy.deepcopy()`

Иногда программисту приходится работать с вложенными друг в друга коллекциями. Например матрица или список списков.

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_m = matrix.copy()
print(matrix, new_m, sep='\n')
matrix[0][1] = 555
print(matrix, new_m, sep='\n')
```

Метод `copy` создал поверхностную копию, копию верхнего уровня. Изменения же вложенных объектов отразится и на оригинале. В таком случае для создания полной копии любой глубины вложенности используют функцию `deepcopy` из модуля `copy`.

```
import copy

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_m = copy.deepcopy(matrix)
print(matrix, new_m, sep='\n')
matrix[0][1] = 555
print(matrix, new_m, sep='\n')
```

Функция рекурсивно обходит все вложенные объекты создавая их копии. Изменения одной коллекции теперь не затрагивают её копию.

Плюсы и минусы создания копии

При работе со списками важно помнить, что сам список как хранитель указателей на объекты занимает место в памяти. Дополнительно занимают память и сами объекты, на которые список указывает. Создание копии приводит к новым затратам памяти, ведь мы создаём новый объект список. Если вы работаете с большими данными, создание копии может быть не лучшей идеей - может не хватить памяти ПК. Кроме того каждая копия требует временных ресурсов на копирование. Прежде чем использовать срезы, копии задумайтесь можно ли решить задачу иначе, экономя время и память.

С другой стороны небольшие списки быстро копируются. И если в вашей задаче важно сохранить оригинал, но нужно модифицировать список для получения результата — копирование вполне допустимо.

Функция len

В финале списков рассмотрим функцию len. На вход она принимает любую коллекцию, в которой можно посчитать количество элементов.



Важно! Функция одинаково работает не только для списков, но и для других коллекций.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
matrix = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
print(len(my_list))
print(len(matrix))
print(len(matrix[1]))
```

Обратите внимание на результат работы функции. Подсчитывается количество элементов первого уровня вложенности. У списка 10 элементов в ячейках с нулевой по девятую. Функция вернула 10.

У матрицы три элемента списка в ячейках с нулевой по вторую. В каждом из вложенных списков по 4 элемента, т.е. суммарно 12. Но функция len вернула число 3 посчитав элементы верхнего уровня.

При обращении к элементу матрицы в ячейке один, т.е. списку [5, 6, 7, 8] функция len возвращает число 4, верно посчитав количество элементов вложенного списка.

Для списков `len` работает за константное время $O(1)$. Внутри объекта списка всегда хранится количество элементов. Функция получает это значение, а не занимается подсчётом.

Задание

Перед вами список и несколько строк кода, которые его меняют. Напишите что вернёт каждая из строк кода. Попробуйте справиться с заданием без запуска кода. У вас 3 минуты.

```
my_list = [2, 4, 6, 2, 8, 10, 12, 14, 16, 18]
print(my_list[2:6:2])
print(my_list.pop())
print(my_list.extend([314, 42]))
print(my_list.sort(reverse=False))
print(my_list)
```

2. Строки, `str`

И снова строки, но на этот раз как массивы символов. Часть рассмотренной для списков информации аналогична и для строки. Например, обращение к элементу строки по индексу в квадратных скобках, срезы строк и т.п. При этом стоит помнить, что строка неизменяема.

Работа со строками как с массивами

Начнём с квадратных скобок.

```
text = 'Hello world!'
print(text[6])
print(text[3:7])
```

Индексы и срезы работают аналогично спискам.

Если необходимо заменить элемент новым, индексы не подойдут. Для этих целей нужен метод `replace`

```
new_txt = text.replace('l', 'L', 2)
print(text, new_txt, sep='\n')
```

Первый аргумент — подстрока, которую нужно заменить.

Второй аргумент — подстрока, на которую нужно заменить.

Третий аргумент — максимальное количество замен. Если его не указывать, будут заменены все совпадения.

Методы count, index, find

Как и у списка, строка поддерживает методы count для подсчёта вхождения и index для поиска элемента. Но у строки появился и новый метод find. Он работает аналогично index. Но если искомая подстрока отсутствует, вместо ошибки возвращает -1.

```
text = 'Hello world!'
print(text.count('l'))
print(text.index('l'))
print(text.find('l'))
print(text.find('z'))
```

Реверс строк

Для разворота строки используется обратный срез, как и в случае со списком.

```
text = 'Hello world!'
print(text[::-1])
```

Форматирование строк

Опытные программисты могут назвать 5-7 способов форматирования строк. Разбирать их все нет смысла. Рассмотрим три “главных” на примерах.

Форматирование через %

Форматирование с использованием символа % является старым способом указания формата. Его вы можете встретить в коде, который писали очень давно. В настоящее время он используется лишь в некоторых модулях для задания формата вывода данных.

```
name = 'Alex'
age = 12
text = 'Меня зовут %s и мне %d лет' % (name, age)
print(text)
```

В строке текста используется знак % с символом типа после него. s — строка, d — число и т.п. После строки указывается символ % и перечисляются переменные. Если переменных больше одной, они заключаются в круглые скобки и разделяются запятой — передаётся кортеж.



Важно! Подробнее про используемые литеры вы можете прочитать по ссылке в материалах лекции. Пока же договоримся не использовать данный стиль форматирования, если это не обязывает конкретный модуль.

Метод format

Метод формат является строковым методом и позволяет соединять заранее заготовленный текст с переменными. Долгое время был основным способом форматирования. До версии Python 3.6, если быть точным.

```
name = 'Alex'
age = 12
text = 'Меня зовут {} и мне {} лет'.format(name, age)
print(text)
```

В строке используются фигурные скобки как место для подстановки значений. Далее для строки вызывается метод format. В качестве аргументов метод получает нужное количество переменных.

f-строка

Начиная с Python 3.7 для форматирования текста используют f-строки. Они работают быстрее, чем старые способы форматирования. А некоторые разработчики языка предлагают сделать их строками по умолчанию в одном из будущих релизов.

f-строки похожи на более короткую и читаемую запись метода формат.

```
name = 'Alex'
age = 12
text = f'Меня зовут {name} и мне {age} лет'
print(text)
```

Перед открывающей кавычкой пишут f — указатель на отформатированную строку. Текст внутри фигурных скобок воспринимается как переменная и на печать выводятся из значения.



Важно! Для печати фигурных скобок используется две фигурные скобки слитно.

```
print(f'{{Фигурные скобки}} и {{name}}')
```

Помимо вывода содержимого переменной можно указать дополнительные символы, влияющие на представление.

Уточнение формата

Существуют различные способы уточнения способа вывода значения переменной.

```
pi = 3.1415
print(f'Число Пи с точностью два знака: {pi:.2f}')
```



```
data = [3254, 4364314532, 43465474, 2342, 462256, 1747]
```



```
for item in data:
    print(f'{item:>10}')
```

```
num = 2 * pi * data[1]
print(f'{num = :_}')
```

После указания имени переменной в фигурных скобках ставится двоеточие — указатель на символы задания формата далее.

- `:.2f` — число пи выводим с точность два знака после запятой
- `:>10` — элементы списка выводятся с выравниванием по правому краю и общей шириной вывода в 10 символов
- `=` — выводим имя переменной, знак равенства с пробелами до и после него и только потом значение.
- `:_` — число разделяется символом подчёркивания для деления на блоки по 3 цифры.

Про эти и другие способы форматирования можно почитать в официальной документации. Ссылка в материалах.

Методы строк

Рассмотрим ещё несколько методов, которые есть только у строк.

Метод `split`

Метод `split` позволяет разбить строку на отдельные элементы в соответствии с разделителем и поместить результат в список.

```
link = 'https://habr.com/ru/users/dzhoker1/posts/'
urls = link.split('/')
print(urls)

a, b, c = input('Введите 3 числа через пробел: ').split()
print(c, b, a)
```

В первом случае мы взяли ссылку и разделили её на отдельные компоненты по символу `/`. Обратите внимание, что между двойным слешем мы получили пустую строку. И вторую пустую строку после последнего слеша.

Во втором примере метод не получил на вход аргументы и деление происходит по пробельным символам. Три переменные до знака равно примут по одному из переданных значений.



Важно! С одной стороны удобно запросить у пользователя три значения в одной строке кода. Но стоит пользователю ошибиться с пробелами и ввести меньше или больше трёх чисел, получим ошибку: `ValueError: too many values to unpack (expected 3)` или `ValueError: not enough values to unpack (expected 3, got 2)`

Один из способов избежать ошибки лишних (но не меньших) данных при распаковке методом `split` — использовать символ распаковки.

```
a, b, c, *_ = input('Введите не менее трёх чисел через пробел: ').split()
```

Переменная подчёркивание благодаря звёздочке, символу упаковки, превращается в список, который заберет значения начиная с четвёртого.

Метод `join`

Метод `join` принимает на вход итерируемую последовательность и соединяет все её элементы в строку, разделяя каждый текстом, к которому применён метод. В некоторой степени `join` противоположен `split`.

```
data = ['https:', '', 'habr.com', 'ru', 'users', 'dzhoker1', 'posts']
url = '/'.join(data)
print(url)
```

К строке `“/”` применили метод, т.е. каждый элемент списка будет разделён слешем. При этом в начале и в конце получившейся строки слеша не будет.

Методы `upper`, `lower`, `title`, `capitalize`

При работе с текстом можно быстро менять строчные буквы на прописные и наоборот.

```
text = 'однажды в СТУДЁНУЮ зИМНЮЮ ПОРУ'  
print(text.upper())  
print(text.lower())  
print(text.title())  
print(text.capitalize())
```

- upper — все символы приводятся к верхнему регистру
- lower — все символы приводятся к нижнему регистру
- title — первый символ каждого слова (разделитель слов - пробел) приводится к верхнему регистру, остальные символы к нижнему
- capitalize — первый символ строки в верхнем регистре, остальные в нижнем

Методы startswith и endswith

Метод startswith проверяет начинается ли строка с заданной подстроки. Метод возвращает истину или ложь. Метод endswith проверяет окончание строки переданной в качестве аргумента подстрокой.

```
text = 'Однажды в студёную зимнюю пору'  
print(text.startswith('Однажды'))  
print(text.endswith('зимнюю', 0, -5))
```

Оба метода помимо подстроки могут принимать параметры start и stop. В этом случае проверка начала либо конца будет проводиться в указанном диапазоне.

Задание

Перед вами строка текста и несколько строк кода, которые её меняют. Напишите что вернёт каждая из строк кода. Попробуйте справиться с заданием без запуска кода. У вас 3 минуты.

```
text = 'Привет, мир!'  
print(text.find(' '))  
print(text.title())  
print(text.split())  
print(f'{text = :>25}')
```

3. Кортеж, tuple

Кортежи — это неизменяемые последовательности, обычно используемые для хранения коллекций разнородных данных. Также используются в случаях, когда требуется неизменяемая последовательность однородных данных. Как и строку кортеж нельзя изменить после создания. При этом кортеж как и список является массивом указателей на объекты любого типа.

Способы создания кортежа

Создать кортеж можно четырьмя способами.

```
a = ()
b1 = 1,
b2 = (1,)
c1 = 1, 2, 3,
c2 = (1, 2, 3)
d = tuple(range(3))
print(a, b1, b2, c1, c2, d, sep='\n')
```

1. Пара круглых скобок создаёт пустой кортеж
2. Один элемент с замыкающей запятой в скобках или без них создаёт кортеж с элементом
3. Несколько элементов разделенных запятыми с замыкающей запятой или в круглых скобках
4. Функция `tuple()`, которой передаётся любой итерируемый объект



Важно! Обратите внимание, что на самом деле кортеж образует запятая, а не круглые скобки. Круглые скобки необязательны, за исключением случая пустого кортежа или когда они необходимы, чтобы избежать синтаксической неоднозначности. Например, `f(a, b, c)` — это вызов функции с тремя аргументами. `f((a, b, c))` — вызов функции с кортежем в качестве единственного аргумента.

Кортежи реализуют все общие операции последовательностей

- Обращение к элементу по индексу
- Срезы
- Методы, которые работают с последовательностью, но не меняют её: `count`, `index`, а также функция `len()`

Задание

Убедимся, что вы сможете провести параллель между кортежами и списками, строками. Перед вами кортеж и несколько строк кода. Напишите, что вернёт каждая из них. Попробуйте справиться с заданием без запуска кода. У вас 3 минуты.

```
my_tuple = (2, 4, 6, 2, 8, 10, 12, 14, 16, 18)
print(my_tuple[2:6:2])
print(my_tuple[-3])
print(my_tuple.count(2))
print(f'{my_tuple = }')
print(my_tuple.index(2, 2))
print(type('text',))
```

4. Словарь, dict

В Python есть изменяемый тип данных словарь. В других языках аналогичная структура данных может называться отображение, mapping, именованный массив, ассоциативный массив, сопоставление и т.п. Словарь представляет набор пар ключ-значение. Ключ — любой неизменяемый тип данных. Значение - любой тип данных. Обращаясь к ключу словаря получают доступ к значению.



Важно! Ключ выступает источником для вычисления хеша. Полученный хеш играет роль числового индекса и указывает на ячейку со значением. В Python вычисление хеша возможно лишь у неизменяемых типов данных. Следовательно, ключ словаря обязан быть неизменяемым объектом. Обычно это строка, целое число (вещественные лучше не использовать, вы же помните о точности округления), либо кортеж или неизменяемое множество.

Способы создания словаря

Для создания словаря есть несколько способов. Например:

- передать набор пар ключ-значение в фигурных скобках,
- использовать знак равенства между ключом и значением,
- передать любую последовательность, каждый элемент которой пара ключ и значение

```
a = {'one': 42, 'two': 3.14, 'ten': 'Hello world!'}
b = dict(one=42, two=3.14, ten='Hello world!')
c = dict([('one', 42), ('two', 3.14), ('ten', 'Hello world!')])
print(a == b == c)
```

Все три способа создают одинаковые словари.



Важно! Вариант b не допускает использования зарезервированных слов. При этом ключи указываются без кавычек, но в словаре становятся ключами типа str.

Добавление нового ключа

Для добавления в существующий словарь новой пары ключ-значение можно использовать обычную операцию присваивания.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4}
my_dict['ten'] = 10
print(my_dict)
```

Доступ к значению словаря

Доступ через квадратные скобки []

Для получения доступа к значению необходимо указать ключ в квадратных скобках после или переменной.

```
TEN = 'ten'
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict['two'])
print(my_dict[TEN])
print(my_dict[1]) # KeyError: 1
```

Ключ может быть указан явно или передам как содержимое переменной, константы. При попытке обратиться к несуществующему ключу получаем ошибку: `KeyError`.

Доступ к ключу позволяет изменять значения. Для этого используем операцию присваивания как и в случае с добавлением новой пары ключ-значение.



Важно! Получить доступ к ключу по значению невозможно.

Доступ через метод `get`

Если ли мы хотим гарантировать отсутствие ошибки `KeyError` при обращении к элементу словаря, можно обратиться к значению через метод `get`, а не квадратные скобки.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.get('two'))
print(my_dict.get('five'))
print(my_dict.get('five', 5))
print(my_dict.get('ten', 5))
```

При обращении к существующему ключу метод `get` работает аналогично доступу к через квадратные скобки. Если обратиться к несуществующему ключу, `get` возвращает `None`. Метод `get` принимает второй аргумент, значение по умолчанию. Если ключ отсутствует в словаре, вместо `None` будет возвращено указанное значение.

Часто используемые методы словарей

Разберем некоторые методы работы со словарями.

Метод setdefault

Метод setdefault похож на get, но отсутствующий ключ добавляется в словарь.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
spam = my_dict.setdefault('five')
print(f'{spam = }\t{my_dict=}')
eggs = my_dict.setdefault('six', 6)
print(f'{eggs = }\t{my_dict=}')
new_spam = my_dict.setdefault('two')
print(f'{new_spam=}\t{my_dict=}')
new_eggs = my_dict.setdefault('one', 1_000)
print(f'{new_eggs=}\t{my_dict=}')
```

При вызове метода с одним аргументом отсутствующий ключ добавляется в словарь. В качестве значения передаётся None. Если указать два аргумента и ключ отсутствует, второй аргумент становится значением ключа и также добавляется в словарь. При обращении к существующему ключу, словарь не изменяется независимо от того указаны один или два аргумента.

Метод keys

Метод keys возвращает объект-итератор dict_keys.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.keys())
for key in my_dict.keys():
    print(key)
```

Обычно объект не используют напрямую. Метод keys применяется в связке с циклом for для перебора ключей словаря.



Важно! Запись цикла `for key in my_dict:` отработает аналогично. По умолчанию словарь возвращает ключи для итерации в цикле.



Внимание! В отличие от списков, кортежей и строк доступ к элементу-значению осуществляется не по индексу, а по ключу. При этом начиная с версии Python 3.7 словарь сохраняет порядок добавления ключей. В

каком порядке ключи были добавлены, в том порядке они будут возвращены в случае итерации по словарю.

Метод values

Метод values похож на keys, но возвращает значения в виде объекта итератора dict_values, а не ключи.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.values())
for value in my_dict.values():
    print(value)
```

Метод items

Если в цикле необходимо работать одновременно с ключами и значениями, как с парами, используют метод items.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.items())
for tuple_data in my_dict.items():
    print(tuple_data)
for key, value in my_dict.items():
    print(f'{key} = {value} before 100 - {100 - value}')
```

Метод возвращает объект итератор dict_items. Если создать цикл for с одной переменной между for и in, получим кортеж из пар элементов — ключа и значения. Обычно используют две переменные в цикле: первая принимает ключ, а вторая значение. Такой подход облегчает чтение кода и позволяют использовать ключ и значение по-отдельности.

Метод popitem

Для удаления пары ключ значение из словаря используют метод popitem.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
spam = my_dict.popitem()
print(f'{spam = }\t{my_dict=}')
eggs = my_dict.popitem()
print(f'{eggs = }\t{my_dict=}')
```

Так как словари сохраняют порядок добавления ключей, удаление происходит справа налево, по методу LIFO. Элементы удаляются в обратном добавлению порядке.



Важно! Если измените значение у существующего ключа, положение ключа в очереди не меняется, он не считается последним добавленным.

Метод pop

Метод pop удаляет пару ключ-значение по переданному ключу.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
spam = my_dict.pop('two')
print(f'{spam = }\t{my_dict=}')
err = my_dict.pop('six') # KeyError: 'six'
err = my_dict.pop()     # TypeError: pop expected at least 1
                        # argument, got 0
```

Если указать несуществующий ключ, получим ошибку KeyError. В отличии от метода pop у списков list, dict.pop вызовет ошибку TypeError. Для удаление последнего элемента нужен метод popitem.

Метод update

Для расширения словаря новыми значениями используют метод update.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
my_dict.update(dict(six=6))
print(my_dict)
my_dict.update(dict([('five', 5), ('two', 42)]))
print(my_dict)
```

На вход метод получает другой словарь в любой из вариаций создания словаря. Если передать существующий ключ, значение будет заменено новым.

Ещё один способ создать словари из нескольких других, который появился в новой версии Python — вертикальная черта.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
new_dict = my_dict | {'five': 5, 'two': 42} | dict(six=6)
print(new_dict)
```

При перезаписи совпадающих ключей приоритет отдаётся словарю, расположенному правее.

Задание

Перед вами словарь и несколько строк кода. Напишите что вернёт каждая из строк. Попробуйте справиться с заданием без запуска кода. У вас 3 минуты.

```
my_dict = {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'ten': 10}
print(my_dict.setdefault('ten', 555))
print(my_dict.values())
print(my_dict.pop('one'))
my_dict['one'] = my_dict['four']
print(my_dict)
```

5. Множества set и frozenset

Ещё одна коллекция из коробки — множества. Множество — набор уникальных неиндексированных элементов. В Python есть два вида множеств: set — изменяемое множество, frozenset — неизменяемое множество. Неизменяемое множество позволяет вычислять хеш и может использоваться там, где разрешён

лишь хешированный тип данных, например в качестве ключа словаря.

```
my_set = {1, 2, 3, 4, 2, 5, 6, 7}
print(my_set)
my_f_set = frozenset((1, 2, 3, 4, 2, 5, 6, 7,))
print(my_f_set)
not_set = {1, 2, 3, 4, 2, 5, 6, 7, ['a', 'b']} # TypeError: unhashable type: 'list'
```

Обратите внимание, что двойка передавалась в множества дважды, но хранится в единственном экземпляре, как один из уникальных элементов



Важно! Элементом множества могут быть только неизменяемые типы данных.

Методы множеств

Рассмотрим некоторые методы множеств на примере изменяемого множества `set`. Все методы, которые не изменяют оригинал, работают аналогично и для множества `frozenset`.

Метод `add`

Метод `add` работает аналогично методу списка `append`, т.е. добавляет один элемент в коллекцию.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
my_set.add(9)
print(my_set)
my_set.add(7)
print(my_set)
my_set.add(9, 10) # TypeError: set.add() takes exactly one
                  argument (2 given)
my_set.add((9, 10))
print(my_set)
```


Если попытаться добавить уже существующий элемент, множество не изменится. При попытке добавить несколько элементов получим ошибку `TypeError`.



Внимание! Если передать в метод `add` неизменяемую коллекцию, например кортеж, коллекция будет добавлена как один целостный объект.

Метод `remove`

Для удаления элемента множества используют метод `remove`.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
my_set.remove(5)
print(my_set)
my_set.remove(10) # KeyError: 10
```

При передаче несуществующего объекта получим ошибку `KeyError`.

Метод `discard`

Метод `discard` работает аналогично `remove` — удаляет один элемент множества.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
my_set.discard(5)
print(my_set)
my_set.discard(10)
```

В отличие от `remove` при попытке удалить несуществующий элемент `discard` не вызывает ошибку. При этом множество не изменяется.

Метод `intersection`

Для получения пересечения множеств, т.е. множества с элементами, которые есть и в левом и в правом множестве используют метод `intersection`

```
my_set = {3, 4, 2, 5, 6, 1, 7}
other_set = {1, 4, 42, 314}
new_set = my_set.intersection(other_set)
print(f'{my_set = }\n{other_set = }\n{new_set = }')
```

Новая версия Python позволяет получить пересечение множеств в следующей записи с использованием символа &

```
my_set = {3, 4, 2, 5, 6, 1, 7}
other_set = {1, 4, 42, 314}
new_set = my_set & other_set
print(f'{my_set = }\n{other_set = }\n{new_set = }')
```



Внимание! Исходные множества при пересечении не изменяются.

Метод union

Для объединения множеств используется метод union.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
other_set = {1, 4, 42, 314}
new_set = my_set.union(other_set)
print(f'{my_set = }\n{other_set = }\n{new_set = }')
new_set_2 = my_set | other_set
print(f'{my_set = }\n{other_set = }\n{new_set_2 = }')
```

На выходе получаем множество уникальных элементов из левого и правого множеств. Более короткая запись объединения возможна при помощи вертикальной черты.

Метод difference

Метод difference удаляет из левого множества элементы правого.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
other_set = {1, 4, 42, 314}
new_set = my_set.difference(other_set)
```

```
print(f'{my_set = }\n{other_set = }\n{new_set = }')
new_set_2 = my_set - other_set
print(f'{my_set = }\n{other_set = }\n{new_set_2 = }')
```

На выходе получаем множество элементов встречающихся только в левом множестве. Более короткая запись возможно при помощи знака минус. Вычитаем из левого элементы правого.

Проверка на вхождение, in

Для проверки входит ли элемент в множество используют зарезервированное слово in.

```
my_set = {3, 4, 2, 5, 6, 1, 7}
print(42 in my_set)
```



Внимание! Слово in позволяет сделать проверку на вхождение и в других коллекциях. Входит ли объект в list, tuple, является ли подстрока частью строки str, встречается ли ключ в словаре. Для list, tuple, str проверка на вхождение работает за линейное время $O(n)$. Для dict, set, frozenset проверка работает за константное время $O(1)$.

Задание

Перед вами множество и несколько строк кода. Напишите что вернёт каждая из строк. Попробуйте справиться с заданием без запуска кода. У вас 3 минуты.

```
my_set = frozenset({3, 4, 1, 2, 5, 6, 1, 7, 2, 7})
print(len(my_set))
print(my_set - {1, 2, 3})
print(my_set.union({2, 4, 6, 8}))
print(my_set & {2, 4, 6, 8})
print(my_set.discard(10))
```

6. Классы bytes и bytearray

Уделим несколько строк лекции неизменяемым байтам и их изменяемой версии — массиву байт. Для отправки информации по каналам связи объекты не подойдут. Даже текст не отправить. А вот пересылать байты — легко.

```
text_en = 'Hello world!'
res = text_en.encode('utf-8')
print(res, type(res))

text_ru = 'Привет, мир!'
res = text_ru.encode('utf-8')
print(res, type(res))
```

Строковый метод `encode` получает в качестве аргумента указание кодировки. На выходе получаем строку байт. Функция `print` возвращает строковое представление байт, сами ячейки памяти с электронами невозможно увидеть невооруженным глазом.

```
b'Hello world!'
b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82,\n\xd0xbc\xd0\xb8\xd1\x80!'
```

Префикс `b` говорит о том, что перед нами не строка, а байты. Если байт может быть представлен как символ, т.е. он есть в семибитной кодировке ASCII, отображается символ. В остальных случаях указывается приставка `\x` и слитно с ней шестнадцатеричное представление байта.

Для получения набора байт можно использовать функцию `bytes`. А если необходимо изменять байт, используя функцию `bytearray`.

```
x = bytes(b'\xd0\x9f\xd1\x80\xd0\xb8')
y = bytearray(b'\xd0\x9f\xd1\x80\xd0\xb8')
print(f'{x = }\n{y = }')
```

В качестве аргумента передаётся строковое представление нужным байт.

Классы байт и массив байт обладают практически всеми методами строк. Кроме того для массива байт доступны методы модификации списка `list`.

7. Вывод

На этой лекции мы:

1. Разобрали, что такое коллекция и какие коллекции есть в Python
2. Изучили работу со списками, как с самой популярной коллекцией
3. Узнали, как работать со строкой в ключе коллекция
4. Разобрали работу с кортежами
5. Узнали, что такое словари и как с ними работать
6. Изучили множества и особенности работы с ними
7. Познакомились с классами байт и массив байт

Краткий анонс следующей лекции

1. Разберёмся с созданием собственных функций в Python
2. Изучим работу встроенных функций

Домашнее задание

1. Поработайте со справочной информацией в Python, функцией `help()`. Попробуйте найти дополнительную информацию о изученных на уроке коллекциях. Если вы плохо читаете на английском, воспользуйтесь любым онлайн переводчиком.
2. Проведите несколько экспериментов с классами `bytes` и `bytearray`. Посмотрите на их поведение как у строки и на списочные методы `bytearray`.

Оглавление

На этой лекции мы	3
Дополнительные материалы к лекции	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	4
1. Создание своих функции	4
Определение функции	5
Что такое pass	6
Аргументы функции	7
Изменяемые и неизменяемые аргументы	8
Возврат значения	9
Значения по умолчанию	11
Изменяемый объект как значение по умолчанию	11
Позиционные и ключевые параметры	12
Параметры args и kwargs	14
Области видимости: global и nonlocal	15
Доступ к константам	17
Анонимная функция lambda	18
Документирование кода функций	19
Задание	20
2. Функции “из коробки”	21
Повторяем map(), filter(), zip()	22
Функция map()	22
Функция filter()	22
Функция zip()	22
Функции max(), min(), sum()	23

Функция max()	23
Функция min()	24
Функция sum()	24
Функции all(), any()	24
Функция all()	24
Функция any()	25
Функции chr(), ord()	26
Функция chr()	26
Функция ord()	26
Функции locals(), globals(), vars()	26
Функция locals()	26
Функция globals()	27
Функция vars()	28
Задание	28
3. Вывод	28

На этой лекции мы

1. Разберёмся с созданием собственных функций в Python.
2. Изучим работу встроенных функций “из коробки”.

Дополнительные материалы к лекции

Встроенные в Python функции: <https://docs.python.org/3/library/functions.html>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрали что такое коллекция и какие коллекции есть в Python.
2. Изучили работу со списками, как с самой популярной коллекцией.
3. Узнали как работать со строкой в ключе коллекции.
4. Разобрали работу с кортежами.
5. Узнали что такое словари и как с ними работать.
6. Изучили множества и особенности работы с ними.
7. Познакомились с классами байт и массив байт.

Термины лекции

- **Функция в программировании, или подпрограмма** — фрагмент программного кода, к которому можно обратиться из другого места программы.
- **Функции высшего порядка** — это функции, которые работают с другими функциями, либо принимая их в виде параметров, либо возвращая их.

Подробный текст лекции

Введение

Функция — фрагмент программного кода, к которому можно обратиться из другого места программы. Функцию стоит представлять как чёрный ящик. В ящик попадают данные, обрабатываются внутри (для пользователя функции не важно как они обрабатываются) и ящик возвращает готовый результат. Подобные функции обеспечивают простоту использования и возможности переносимости и переиспользования в других частях проекта и даже в других проектах.

В Python можно создавать свои функции, использовать встроенные функции интерпретатора, а также работать с функциями из модулей и пакетов стандартной библиотеки и из устанавливаемых дополнений. На этой лекции подробно поговорим про самописные функции и функции “из коробки”.

1. Создание своих функции

Как и всё в Python функция является объектом. Можно сказать, что питоновская функция это функция высшего порядка. Она может работать с другими функциями, либо принимая их в виде параметров, либо возвращая их. Проще говоря, функцией высшего порядка называется такая функция, которая принимает функцию-объект как аргумент или возвращает функцию-объект в виде выходного значения.

Разберёмся в понятиях “вызываем” и “передаём” на уже знакомом примере кода.

```
a = 42
print(type(a), id(a))
print(type(id))
```

Функция `print` вызывается с двумя аргументами - функциями. Каждая из переданных в качестве аргументов функций: `type` и `id` так же вызываются с переменной `a` в качестве аргумента.

Во втором случае функция `type` вызывается с функцией `id` в качестве аргумента. При этом у `id` отсутствуют круглые скобки после имени. Мы не вызываем её, а передаём как объект.

Ещё один пример передачи функции ниже.

```
very_bad_programming_style = sum
print(very_bad_programming_style([1, 2, 3]))
```

Передали в переменную встроенную функцию `sum`. Теперь переменную можно вызывать как функцию суммирования.

Итого. Наличие круглых скобок после имени функции с аргументами или без них внутри скобок — **вызов** функции. Имя функции без скобок — **передача** функции как объекта.

Определение функции

Для определения собственной функции используется зарезервированное слово `def`. Далее указывается имя функции, круглые скобки с параметрами при необходимости и двоеточием. Со следующей строки описывается тело функции как

вложенный блок, т.е. с 4 отступами для каждой строки тела.

```
def my_func():  
    pass
```

Определили функцию под именем my_func, которая не принимает аргументы и ничего не делает.



PEP-8! Имена функций записываются в стиле snake_case как и имена переменных

Что такое pass

Похоже мы впервые встретили слово pass. Это зарезервированное слово, которое ничего не делает. Используется в тех местах, где должен быть код для верной работы программы, но его пока нет. Например мы не можем создать функцию без тела. Нужна как минимум одна строка. В ней мы и написали pass.



Внимание! Не злоупотребляйте использованием pass. Делать десятки заготовок функций и классов с pass на будущее - не лучшая привычка. Создавайте код по мере того как он вам нужен.

И сразу пару антипримеров использования pass которые выдают в программисте новичка, радостно вставляющего новое слово повсюду в коде.

Плохо:

```
if a != 5:  
    pass  
else:  
    a += 1
```

Уже лучше:

```
if a == 5:  
    a += 1  
else:  
    pass
```

Отлично. Ничего лишнего:

```
if a == 5:
    a += 1
```

Аргументы функции

Рассмотрим функцию с параметрами, т.е. принимающую на вход значения. Вспомним как в школе решали квадратные уравнения. Заодно разберём особенности создания функций в Python.

```
def quadratic_equations(a: int | float, b: int | float, c: int | float) -> tuple[float, float] | float | str:
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 * a)
    elif d == 0:
        return -b / (2 * a)
    else:
        return 'Нет решений'

print(quadratic_equations(2, -3, -9))
```

Во первых воспользовались аннотацией типов. Указали, что на вход ожидаем три значения целого или вещественного типа. Далее после стрелки указано, что функция может вернуть кортеж с парой вещественных значений, т.е. два корня уравнения, либо одно число когда дискриминант равен нулю, либо строку — фразу нет решений.



Внимание! Обратите внимание, что смешивать текст и числа в выводе — не лучшая идея. Тот, кто будет использовать функцию будет вынужден делать проверки на тип возвращаемого значения. Логичнее возвращать None, если уравнение не имеет решений. В таком случае первая строка будет записана как:


```
def quadratic_equations(a: int | float, b: int | float, c: int | float) -> tuple[float, float] | float | None:
```

А в последней строке вернём ничего:

```
return None
```

Кстати без указания типов первая строка могла быть записана как:

```
def quadratic_equations(a, b, c):
```

 **PEP-8!** Обратите внимание на пустые строки между функцией и её вызовом. Рекомендуется оставлять по 2 пустых строки как после, так и до функции.

В нашем примере мы указали 3 позиционных параметра. При вызове функции значения попадают в соответствующие позиции переменные. При попытке передать отличное от указанного в описании функции число аргументов получим ошибку `TypeError`.

```
print(quadratic_equations(2, -3)) # TypeError:
quadratic_equations() missing 1 required positional argument: 'c'
```

Изменяемые и неизменяемые аргументы

При передаче аргументов в функцию стоит помнить изменяемого типа объект или нет. От этого зависит поменяем мы оригинал или он останется неизменным. Пример работы с неизменяемыми переменными.

```
def no_mutable(a: int) -> int:
    a += 1
    print(f'In func {a = }') # Для демонстрации работы, но не
    # для привычки принтить из функции
    return a

a = 42
print(f'In main {a = }')
z = no_mutable(a)
print(f'{a = }\t{z = }')
```

Попытка изменить содержимое переменной внутри функции не привела к изменению одноимённой переменной вне функции. Подробнее об областях видимости далее в лекции.

А пока пример работы с изменяемыми объектами.

```
def mutable(data: list[int]) -> list[int]:
    for i, item in enumerate(data):
        data[i] = item + 1
    print(f'In func {data = }') # Для демонстрации работы, но не
    # для привычки принтить из функции
    return data

my_list = [2, 4, 6, 8]
print(f'In main {my_list = }')
new_list = mutable(my_list)
print(f'{my_list = }\t{new_list = }')
```

Изменение списка внутри функции привело к изменению списка вне функции.



Важно! В Python аргументы передаются внутри функции по ссылке на объект. Следовательно изменяемый объект можно поменять. А при попытке изменить неизменяемый создадим новый объект либо получим ошибку.

Возврат значения

Как вы уже заметили внутри функции на печать ничего не выводится. Мы не используем `print` в теле, а выводим на печать результат работы функции. Такой подход предпочтительнее. Ведь результат можно сохранить в переменную для дальнейшей работы с ним.

Для возврата результатов используется зарезервированное слово `return`. Указанные после слова объекты возвращаются как результат работы функции.

- Если указан один объект — возвращается именно этот объект.
- Если указано несколько значений через запятую, возвращается кортеж с перечисленными значениями
- Если ничего не указано после `return` — возвращается `None`

После выполнения строки с командой `return` работа функции завершается, даже если это не последняя строка в теле функции.

```
def quadratic_equations(a, b, c):
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
    if d == 0:
        return -b / (2 * a)
    return None
```

Мы убрали else, т.к. последний возврат будет выполнен только в том случае, когда не сработала ни одно условие выше и следовательно не получилось выйти из функции через другой return. По этой же причине заменили elif на if.

Неявный return

Иногда функции не возвращают значения. Точнее нет явного возврата. В этом случае Python “мысленно” дописывает в качестве последней строки функции return None. Немного изменим прошлый пример и посмотрим на новый результат.

```
def no_return(data: list[int]):
    for i, item in enumerate(data):
        data[i] = item + 1
    print(f'In func {data = }') # Для демонстрации работы, но не
для привычки принтить из функции

my_list = [2, 4, 6, 8]
print(f'In main {my_list = }')
new_list = no_return(my_list)
print(f'{my_list = }\t{new_list = }')
```

Функция состоит из 4 строк кода. Можно представить, что в 5-й автоматически дописался возврат None. Именно по этой причине в переменной new_list вместо списка содержится None.



Важно! Если функция ничего не возвращает, значит она возвращает ничего — None. Теперь можно изменить функцию поиска корней квадратного уравнения и сделать короче.

```
def quadratic_equations(a, b, c):
```

```

d = b ** 2 - 4 * a * c
if d > 0:
    return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
if d == 0:
    return -b / (2 * a)

```

Значения по умолчанию

Функция может содержать значения по умолчанию для своих параметров. Например:


```

def quadratic_equations(a, b=0, c=0):
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
    if d == 0:
        return -b / (2 * a)

print(quadratic_equations(2, -9))

```

Переменная `a` должна быть передана в обязательном порядке. Если не передать 2-й и/или 3-й аргумент, в переменные попадут нули как значения по умолчанию.

 **PEP-8!** Для указания значения по умолчанию используется знак равенства. До и после такого равно пробелы не ставятся.

Изменяемый объект как значение по умолчанию

В качестве значения по умолчанию нельзя указывать изменяемые типы: списки, словари и т.п. Это приведёт к неожиданным результатам:

```
def from_one_to_n(n, data=[]):
    for i in range(1, n + 1):
        data.append(i)
    return data

new_list = from_one_to_n(5)
print(f'{new_list = }')
other_list = from_one_to_n(7)
print(f'{other_list = }')
```

`other_list` содержит в себе и новую последовательность и ту, которая была в списке `new_list`. Связано это с тем, что значение по умолчанию задаётся один раз при создании функции. Каждый вызов — работа со списком `data` и его очередное изменение.

Если в качестве значения по умолчанию нужен изменяемый тип данных, используют особый приём с `None`

```
def from_one_to_n(n, data=None):
    if data is None:
        data = []
    for i in range(1, n + 1):
        data.append(i)
    return data
```

Если изменяемый объект не передан, он создаётся по новой для каждого вызова функции.

Позиционные и ключевые параметры

Пришло время поговорить о позиционных и ключевых параметрах функции. Начнём с общего синтаксиса.

```
def func(positional_only_parameters, /, positional_or_keyword_parameters, *,
keyword_only_parameters):
    pass
```

При указании параметров функции вначале идут позиционные параметры. При вызове функции передаются значения без указания имени переменной-аргумента. Косая черта не является переменной. Это символ разделитель. После неё могут

идти как позиционные, так и ключевые параметры. Далее символ разделитель звёздочка указывает только на ключевые параметры.



Важно! Косая черта и звёздочка одновременно или по отдельности могут отсутствовать при определении функции.

Разберем передачу аргументов по позиции и по имени на примерах.

- **Пример обычной функции:**

```
def standard_arg(arg):  
    print(arg)    # Принтим для примера, а не для привычки  
  
standard_arg(42)  
standard_arg(arg=42)
```

Функция может принимать значения по позиции и по ключу. Мы явно указали имя переменной.

- **Пример только позиционной функции:**

```
def pos_only_arg(arg, /):  
    print(arg)    # Принтим для примера, а не для привычки  
  
pos_only_arg(42)  
pos_only_arg(arg=42)    # TypeError: pos_only_arg() got some  
positional-only arguments passed as keyword arguments: 'arg'
```

Теперь функция принимает только позиционные параметры.

- **Пример только ключевой функции:**

```
def kwd_only_arg(*, arg):  
    print(arg)    # Принтим для примера, а не для привычки  
  
kwd_only_arg(42)  
kwd_only_arg(arg=42)
```

А теперь наоборот, можем принимать только значения по ключу.

- **Пример функции со всеми вариантами параметров:**

```
def combined_example(pos_only, /, standard, *, kwd_only):
```


```

    print(pos_only, standard, kwd_only)  # Принтим для примера, а
не для привычки

combined_example(1, 2, 3)  # TypeError: combined_example() takes
2 positional arguments but 3 were given
combined_example(1, 2, kwd_only=3)
combined_example(1, standard=2, kwd_only=3)
combined_example(pos_only=1, standard=2, kwd_only=3)  #
TypeError: combined_example() got some positional-only arguments
passed as keyword arguments: 'pos_only'

```

И наконец пример со всеми возможными вариантами параметров.

 **Важно!** Если функция принимает несколько ключевых параметров, порядок передачи аргументов может отличаться.

```

def triangulation(*, x, y, z):
    pass

triangulation(y=5, z=2, x=11)

```

Параметры args и kwargs

Отдельно разберём пару особых параметров. Звёздочка args и две звёздочки kwargs.

Важно! Python в первую очередь смотрит на звёздочки, а не на имя переменной. Но среди разработчиков приняты имена args и kwargs. Они делают код привычным, т.е. повышают читаемость. Не используйте другие переменные.

Начнём с *args:

```

def mean(args):
    return sum(args) / len(args)

print(mean([1, 2, 3]))
print(mean(1, 2, 3))  # TypeError: mean() takes 1 positional


```

argument but 3 were given

Функция может принять лишь один аргумент. В случае со списком проблем нет. Но если перечислить элементы через запятую и не указать скобки - не передать кортеж, получим ошибку.

```
def mean(*args):  
    return sum(args) / len(args)  
  
print(mean(*[1, 2, 3]))  
print(mean(1, 2, 3))
```


Теперь функция принимает любое число позиционных аргументов. Переменная args превращается в кортеж. Можно сказать, что звёздочка упаковала все позиционные аргументы в один кортеж.

 **Внимание!** При вызове функции со списком перед квадратными скобками добавили звёздочку. Смысл её противоположный. Мы распаковали список. Каждый элемент был передан в функцию по отдельности.

Параметр ****kwargs** работает аналогично, но принимает ключевые параметры и возвращает словарь.

```
def school_print(**kwargs):  
    for key, value in kwargs.items():  
        print(f'По предмету "{key}" получена оценка {value}')
```

```
school_print(химия=5, физика=4, математика=5, физра=5)
```

 **Важно!** Благодаря кодировке utf-8 мы смогли передать в функцию переменные на русском языке.

При написании своих функций стоит помнить о возможности сочетать позиционные и ключевые переменные, специальные символы разделители / и *, а также переменные *args и **kwargs.

Области видимости: global и nonlocal

Хорошая функция работает как чёрный ящик. Использует только переданные ей значения и возвращает ответ. Но в Python функции могут обращаться к внешним переменным без явной передачи в качестве аргумента.

В Python есть несколько областей видимости:

- локальная — код внутри самой функции, т.е. переменные заданные в теле функции.
- глобальная — код модуля, т.е. переменные заданные в файле py содержащем функцию.
- не локальная — код внешней функции, исключающий доступ к глобальным переменным.

Разберем на примерах.

- **Локальные переменные:**

```
def func(y: int) -> int:
    x = 100
    print(f'In func {x = }')  # Для демонстрации работы, но не
    для привычки принтить из функции
    return y + 1

x = 42
print(f'In main {x = }')
z = func(x)
print(f'{x = }\t{z = }')
```

Переменная x в теле функции и переменная x в основном коде - две разные переменные. Локальная область видимости функции создала свою переменную. Попробуем для эксперимента заменить строку с иском на `x += 100` В результате получаем ошибку `UnboundLocalError: local variable 'x' referenced before assignment`. Функция не смогла увеличить 42 на 100, т.к. переменные лишь для нас выглядят одинаково. Чёрный ящик не увидел x без его явной передачи в функцию.

- **Глобальные переменные:**

```
def func(y: int) -> int:
    global x
    x += 100
    print(f'In func {x = }')  # Для демонстрации работы, но не
    для привычки принтить из функции
    return y + 1
```

```
x = 42
print(f'In main {x = }')
z = func(x)
print(f'{x = }\t{z = }')
```

Теперь переменная `x` в теле функции объявлена как глобальная. Мы получили доступ к внешнему `x` со значением 42 и смогли его увеличить. Изменение затронуло как внешний, так и внутренний `x`.

Важно! Не стоит злоупотреблять командой `global`. В 9 из 10 случаев переменную стоит передать как аргумент в функцию и вернуть ответ.

- **Не локальные переменные:**

```
def main(a):
    x = 1

    def func(y):
        nonlocal x
        x += 100
        print(f'In func {x = }')  # Для демонстрации работы, но
        не для привычки принтить из функции
        return y + 1

    return x + func(a)

x = 42
print(f'In main {x = }')
z = main(x)
print(f'{x = }\t{z = }')
```

Функция `func` вложена в функцию `main`. Благодаря команде `nonlocal` мы смогли получить доступ к `x = 1`. В результате внутри `func` `x` увеличился до 101. В отличие от команды `global`, мы не смогли увидеть внешний `x = 42`. `nonlocal` позволяет заглянуть на верхний уровень вложенности, но не выходить на глобальные переменные модуля.

Доступ к константам

Один из случаев когда обращение из тела функции к глобальной переменной считается нормальным — доступ к константам.

```
LIMIT = 1_000

def func(x, y):
    result = x ** y % LIMIT
    return result

print(func(42, 73))
```

Константа `LIMIT` является глобальной. При обращении к ней из функции производится поиск в локальной области, т.е. в теле функции. Далее поиск переходит на уровень выше, в глобальную область видимости модуля. Чтение значений констант внутри функции будет работать без ошибок.

Анонимная функция `lambda`

Анонимные функции, они же лямбда функции — синтаксический сахар для обычных питоновских функций с рядом ограничений. Они позволяют задать функцию в одну строку кода без использования других ключевых слов.

Рассмотрим пример обычной функции и её аналог в виде лямбды для проведения параллели.

```
def add_two_def(a, b):
    return a + b

add_two_lambda = lambda a, b: a + b

print(add_two_def(42, 3.14))
print(add_two_lambda(42, 3.14))
```

После зарезервированного слова `lambda` перечисляются параметры функции через запятую. Далее ставят двоеточие и указывают значение, которое необходимо вернуть без добавления `return`. Функция записывается в одну строку.



Важно! С точки зрения разработки присваивание анонимной функции имени является неверным. Лямбды не должны использоваться для подобных программных решений. Аналогичное предупреждение есть и в PEP-8.

Обычно лямбды используют там, где однократно нужна функция и нет смысла заводить определение через `def`

```
my_dict = {'two': 2, 'one': 1, 'four': 4, 'three': 3, 'ten': 10}
s_key = sorted(my_dict.items())
s_value = sorted(my_dict.items(), key=lambda x: x[1])
print(f'{s_key = }\n{s_value = }')
```

В первом случае словарь сортируется по ключам, т.е. по алфавиту. Во втором благодаря лямбде указали сортировку по второму (индексация начинается с нуля) элементу, т.е. по значению.

Документирование кода функций

Несколько слов о документировании функций. Начнём с того, что документация обычно пишется на английском языке, как универсальном для программистов из любой страны. Вполне допустим и родной язык, если проект локальный. Но лучше воспользоваться онлайн переводчиком и сразу привыкнуть к документированию на английском.

```
def max_before_hundred(*args):
    """Return the maximum number not exceeding 100."""
    m = float('-inf')
    for item in args:
        if m < item < 100:
            m = item
    return m

print(max_before_hundred(-42, 73, 256, 0))
```

Пояснения к однострочной строке документации

- Тройные кавычки используются, даже если строка помещается на одной строке. Это позволяет легко расширить его позже.
- Закрывающие кавычки находятся на той же строке, что и открывающие. Это выглядит лучше для однострочников.
- Нет пустой строки ни до, ни после строки документации.
- Строка документации — это фраза, заканчивающаяся точкой. Он описывает действие функции или метода как команду
- Однострочная строка документации не должна повторять параметры функции.



Внимание! В программе использована переменная, а точнее константа “минус бесконечность” `float('-inf')`. Это особая форма представления бесконечности в памяти интерпретатора, аналогичная бесконечности из модуля `math`.

Если описание функции подразумевает больше подробностей, после первой строки документации оставляют одну пустую. Далее в несколько строк даётся всё необходимое описание. Закрывающие кавычки ставятся на отдельной строке, без текста.

```
def max_before_hundred(*args):  
    """Return the maximum number not exceeding 100.  
  
    :param args: tuple of int or float numbers  
    :return: int or float number from the tuple args  
    """  
    ...
```

Подобная запись автоматически помещает текст в переменную `__doc__`. Описание функции можно будет получить через вызов справки `help` с передачей функции в качестве аргумента.

```
help(max_before_hundred)
```


Задание

Перед вами функция и её вызов. Найдите три ошибки и напишите о них в чат. У вас три минуты.

```
def func(a=0.0, /, b=0.0, *, c=0.0):  
    """func(a=0.0: int | float, /, b=0.0: int | float, *, c=0.0:  
    int | float) -> : int | float"""  
    if a > c:  
        return a  
    if a < c:  
        return c  
    return  
  
print(func(c=1, b=2, a=3))
```

2. Функции “из коробки”

В Python есть ряд встроенных функций. Они доступны всегда, без импортов и других подготовительных операций. Перечислим их в алфавитном порядке:

abs(), aiter(), all(), any(), anext(), ascii(), bin(), bool(), breakpoint(), bytearray(), bytes(), callable(), chr(), classmethod(), compile(), complex(), delattr(), dict(), dir(), divmod(), enumerate(), eval(), exec(), filter(), float(), format(), frozenset(), getattr(), globals(), hasattr(), hash(), help(), hex(), id(), input(), int(), isinstance(), issubclass(), iter(), len(), list(), locals(), map(), max(), memoryview(), min(), next(), object(), oct(), open(), ord(), pow(), print(), property(), range(), repr(), reversed(), round(), set(), setattr(), slice(), sorted(), staticmethod(), str(), sum(), super(), tuple(), type(), vars(), zip().

Часть функций мы уже разбирали на прошлых лекциях. Ещё часть разберём сегодня. О некоторых функциях поговорим на следующих лекциях курса.



Важно! Не используйте имена встроенных функций в качестве имён переменных.



PEP-8! Если очень хочется, добавляйте к имени переменной символ подчёркивания. Не `sum`, а `sum_`. Не `max`, а `max_`

Повторяем `map()`, `filter()`, `zip()`

Повторим уже знакомые вам по прошлым курсам функции для обработки последовательностей.

Функция `map()`

`map(function, iterable)` — принимает на вход функцию и последовательность. Функция применяется к каждому элементу последовательности и возвращает `map` итератор.

```
texts = ["Привет", "ЗДОРОВА", "привеТствую"]
res = map(lambda x: x.lower(), texts)
print(*res)
```

В качестве функции использовали лямбду для вызова метода `lower` у каждого из переданных объектов. Объект итератор `res` был распакован в функцию `print` через символ “звёздочка”.

- **Функция `filter()`**

`filter(function, iterable)` — принимает на вход функцию и последовательность. Если функция возвращает истину, элемент остаётся в последовательности. Как и `map` возвращает объект итератор.

```
numbers = [42, -73, 1024]
res = tuple(filter(lambda x: x > 0, numbers))
print(res)
```

Лямбда фильтрует элементы больше нуля. Функция `tuple` преобразует итератор к кортежу с положительными числами.

- **Функция zip()**

zip(*iterables, strict=False) — принимает несколько последовательностей и итерируется по ним параллельно.

Если передать ключевой аргумент `strict=True`, вызовет ошибку `ValueError` в случае разного числа элементов в каждой из последовательностей.

```
names = ["Иван", "Николай", "Пётр"]
salaries = [125_000, 96_000, 109_000]
awards = [0.1, 0.25, 0.13, 0.99]
for name, salary, award in zip(names, salaries, awards):
    print(f'{name} заработал {salary:.2f} денег и премию {salary * award:.2f}')
```

Последовательно получаем имена, зарплату и процент премии из каждого списка. Итерация идёт слева направо.

Функции max(), min(), sum()

Несколько слов о функциях поиска максимума, минимума и подсчёта суммы.

- **Функция max()**

`max(iterable, *, key, default)` или `max(arg1, arg2, *args[, key])`

Функция принимает на вход итерируемую последовательность или несколько позиционных элементов и ищет максимальное из них. Ключевой параметр `key` указывает на то, какие элементы необходимо сравнить, если объект является сложной структурой. Отдельно параметр `default` используется для возврата значения, если на вход передана пустой итератор.

```
lst_1 = []
lst_2 = [42, 256, 73]
lst_3 = [("Иван", 125_000), ("Николай", 96_000), ("Пётр", 109_000)]
print(max(lst_1, default='empty'))
print(max(*lst_2))
print(max(lst_3, key=lambda x: x[1]))
```

В первом случае передана пустая последовательность и функция вернула строку `empty`.

Во втором — распаковали список и нашли максимальное число.

В третьем ищем максимальное среди трёх кортежей по элементу с индексом один, т.е. по числу.

- **Функция min()**

`min(iterable, *, key, default)]` или `min(arg1, arg2, *args[, key])`

Функция работает аналогично `max`, но ищет минимальный элемент.

```
lst_1 = []
lst_2 = [42, 256, 73]
lst_3 = [("Иван", 125_000), ("Николай", 96_000), ("Пётр", 109_000)]
print(min(lst_1, default='empty'))
print(min(*lst_2))
print(min(lst_3, key=lambda x: x[1]))
```

- **Функция sum()**

`sum(iterable, /, start=0)`

Функция принимает объект итератор и подсчитывает сумму всех элементов. Ключевой аргумент `start` задаёт начальное значение для суммирования.

```
my_list = [42, 256, 73]
print(sum(my_list))
print(sum(my_list, start=1024))
```

Функции `all()`, `any()`

- **Функция all()**

`all(iterable)`

Функция возвращает истину, если все элементы последовательности являются истиной. На Python создание функции `all` выглядело бы так:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

Функция `all` обычно применяется с результатами каких-то вычислений, которые должны быть истинными или ложными.

```
numbers = [42, -73, 1024]
if all(map(lambda x: x > 0, numbers)):
    print('Все элементы положительные')
else:
    print('В последовательности есть отрицательные и/или нулевые элементы')
```

Функция `map` заменила числа на `True` и `False`, далее `all` проверила все ли элементы больше нуля или есть как минимум один не более нуля.

- **Функция `any()`**

`any(iterable)`

Функция возвращает истину, если хотя бы один элемент последовательности является истиной. На Python создание функции `any` выглядело бы так:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Функция `any` работает аналогично `all`. Но если `all` можно представить как `if` с цепочкой `and`, то `any` — это `if` с цепочкой `or`.

Функция `map` заменила числа на `True` и `False`, далее `all` проверила все ли элементы больше нуля или есть как минимум один не более нуля.

```
numbers = [42, -73, 1024]
if any(map(lambda x: x > 0, numbers)):
    print('Хотя бы один элемент положительный')
else:
    print('Все элементы не больше нуля')
```



Важно! Все перечисленные выше функции имеют линейную асимптотику $O(n)$, т.е. функция проходит последовательность от начала до конца прежде чем вернуть результат.

Функции chr(), ord()

Рассмотрим пару функций для работе со символами и их кодами в Юникод.

- **Функция chr()**

chr(integer)

Функция возвращает строковый символ из таблицы Юникод по его номеру. Номер - целое число от 0 до 1_114_111.

```
print(chr(97))
print(chr(1105))
print(chr(128519))
```

- **Функция ord()**

ord(char)

Функция принимает один символ и возвращает его код в таблице Юникод.

```
print(ord('a'))
print(ord('a'))
print(ord('😊'))
```

Функции ord и chr выполняют противоположные действия.

Функции locals(), globals(), vars()

И в финале несколько функций о переменных и областях видимости.

- **Функция locals()**


Функция возвращает словарь переменных из локальной области видимости на момент вызова функции.

```
SIZE = 10

def func(a, b, c):
    x = a + b
    print(locals())
    z = x + c
    return z
```

```
func(1, 2, 3)
```

Функция вернула словарь с переменными a, b, c, x и их значениями. Константа SIZE не попала в вывод, т.к. не входит в локальную область функции. Так же в словаре отсутствует переменная z. Она была впервые создана после вызова функции locals.

 **Важно!** Python игнорирует попытки обновления словаря locals. Для изменения значений переменных надо обращаться к ним напрямую.

- **Функция globals()**

Функция возвращает словарь переменных из глобальной области видимости, т.е. из пространства модуля.


```
SIZE = 10

def func(a, b, c):
    x = a + b
    print(globals())
    z = x + c
    return z

print(globals())
print(func(1, 2, 3))
```

Функция не сохраняет в словаре локальные переменные функций, даже если будет вызвана из тела функции.

В словаре от globals содержатся и дандер переменные модуля. Они нужны Python для правильной работы кода.

 **Внимание!** Если вызвать функцию locals() из основного кода модуля, а не из функции, результат будет аналогичен работе функции globals()

```
x = 42
glob_dict = globals()
glob_dict['x'] = 73
print(x)
```

В отличие от locals словарь globals позволяет изменить значение переменной.

- **Функция vars()**

Функция без аргументов работает аналогично функции locals(). Если передать в vars объект, функция возвращает его атрибут __dict__. А если такого атрибута нет у объекта, вызывает ошибку TypeError.

```
print(vars(int))
```

Получили все дандер методы класса int

Задание

Перед вами список и три операции с ним. Напишите что выведет каждая из строк по вашему мнению, не запуская код. У вас 3 минуты.

```
data = [25, -42, 146, 73, -100, 12]
print(list(map(str, data)))
print(max(data, key=lambda x: -x))
print(*filter(lambda x: not x[0].startswith('__'),
globals().items()))
```

3. Вывод

На этой лекции мы:

1. Разобрались с созданием собственных функций в Python
2. Изучили работу встроенных функций “из коробки”.

Краткий анонс следующей лекции

1. Разобрать решения задач в одну строку
2. Изучить итераторы и особенности их работы
3. Узнать о генераторных выражениях и генераторах списков, словарей, множеств

4. Разобрать создание собственных функций генераторов.

Домашнее задание

1. Поработайте со справочной информацией в Python, функцией `help()`. Попробуйте найти информацию об изученных на уроке функциях “из коробки”. Почитайте описание тех функций, которые не рассматривались на уроке. Если вы плохо читаете на английском, воспользуйтесь любым онлайн переводчиком.

Оглавление

На этой лекции мы	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
1. Однострочники	4
Полезные однострочники	4
Обмен значения переменных	4
Распаковка коллекции	4
Распаковка коллекции с упаковкой “лишнего”, упаковка со звёздочкой	5
Распаковка со звёздочкой	6
Множественное присваивание	6
Множественное сравнение	7
Плохие однострочники	8
Задание	8
2. Итераторы	9
Функции <code>iter()</code> и <code>next()</code>	9
Функция <code>iter</code>	9
Функция <code>next</code>	11
Задание	12
3. Генераторы	12
Комбинации <code>for</code> и <code>if</code> в генераторах и выражениях	13
Допустимые размеры однострочника	14
List comprehensions	15
Генераторные выражения или генерация списка	16
Set comprehensions	16
Dict comprehensions	17
Задание	18
4. Создание функции генератора	18

Команда yield	19
Функции iter и next для генераторов	20
Задание	20
Вывод	20

На этой лекции мы

1. Разберем решения задач в одну строку
2. Изучим итераторы и особенности их работы
3. Узнаем о генераторных выражениях и генераторах списков, словарей, множеств
4. Разберем создание собственных функций генераторов.

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с созданием собственных функций в Python
2. Изучили работу встроенных функций “из коробки”

Термины лекции

- **Итератор** — это объект, представляющий поток данных; объект возвращает данные по одному элементу за раз. Итератор Python должен поддерживать метод с именем `__next__()`, который не принимает аргументов и всегда возвращает следующий элемент потока.
- **Генератор** — это объект, который сразу при создании не вычисляет значения всех своих элементов. Он хранит в памяти только последний вычисленный элемент, правило перехода к следующему и условие, при котором выполнение прерывается. Вычисление следующего значения происходит

лишь при выполнении метода `next()`. Предыдущее значение при этом теряется.

- **Факториал** — функция, определённая на множестве неотрицательных целых чисел. Название происходит от лат. *factorialis* — действующий, производящий, умножающий; обозначается $n!$, произносится эн факториал. Факториал натурального числа n определяется как произведение всех натуральных чисел от 1 до n включительно.

Подробный текст лекции

1. Однострочники

Python позволяет разработчикам решать задачи быстрее, чем с использованием других языков. Одна из причин — возможность писать меньше кода для получения результата.

Полезные однострочники

Рассмотрим несколько примеров кода в одну строку, которые упрощают жизнь.

- Обмен значения переменных

Начнём с классического обмена значений переменных.

```
a, b = b, a
```

Мы поменяли местами содержимое переменных без создания дополнительных, в одну строку.

- Распаковка коллекции

Рассмотрим другие варианты упаковки и распаковки значений.

```
a, b, c = input("Три символа: ")
print(f'{a=} {b=} {c=}')

```

Как вы помните функция `input` возвращает строку `str`. Указав не одну переменную, а три мы распаковываем строку из 3-х символов в три отдельные переменные.

```
a, b, c = ("один", "два", "три",)
print(f'{a=} {b=} {c=}')
```

Аналогичным образом можно распаковать кортеж из трёх элементов в три переменные. Со списком `list`, множеством `set` и прочими коллекциями будет работать аналогично.

```
a, b, c = {"один", "два", "три", "четыре", "пять"}
print(f'{a=} {b=} {c=}')
```

ValueError: too many values to unpack (expected 3)

Если количество переменных слева от равенства не совпадает с количеством элементов последовательности Python вернёт ошибку.

- Распаковка коллекции с упаковкой “лишнего”, упаковка со звёздочкой

Для упаковки может применяться символ “звёздочка” перед именем переменной. Такая переменная превратиться в список и соберёт в себя все значения, не поместившиеся в остальные переменные.

```
data = ["один", "два", "три", "четыре", "пять", "шесть", "семь",
]
a, b, c, *d = data
print(f'{a=} {b=} {c=} {d=}')
```

```
a, b, *c, d = data
print(f'{a=} {b=} {c=} {d=}')
```

```
a, *b, c, d = data
print(f'{a=} {b=} {c=} {d=}')
```

```
*a, b, c, d = data
print(f'{a=} {b=} {c=} {d=}')
```

Элементы коллекции попадают в переменные в зависимости от того, какая из переменных отмечена звёздочкой.



Важно! Звёздочкой можно отметить только одну переменную из перечня.

Если нам нужна часть данных в переменных, а упакованный список в дальнейших расчётах не участвует, в качестве переменной используют подчеркивание.

```
link = 'https://docs.python.org/3/faq/programming.html#how-can-i-pass-optional-or-keyword-parameters-from-one-function-to-another'
prefix, *_ , suffix = link.split('/')

```

- Распаковка со звёздочкой

Ещё один способ применения звёздочки — распаковка элементов коллекции.

Длинный вариант вывода элементов последовательности в одну строку с разделителем табуляцией:

```
data = [2, 4, 6, 8, 10, ]
for item in data:
    print(item, end='\t')

```

И аналогичная операция в одну строку с распаковкой:

```
data = [2, 4, 6, 8, 10, ]
print(*data, sep='\t')

```

- Множественное присваивание

Если несколько переменных должны получить одинаковые значение, можно объединить несколько строк в одну.

```
a = b = c = 0
a += 42
print(f'{a=} {b=} {c=}')

```

Подобная запись допустима только с неизменяемыми типами данных. В противном случае изменение одной переменной приведёт к изменению и других.

```
a = b = c = {1, 2, 3}
a.add(42)
print(f'{a=} {b=} {c=}')

```


Другой вариант множественного присваивания похож на обмен переменных местами.

```
a, b, c = 1, 2, 3
print(f'{a=} {b=} {c=}')
```

Число элементов в левой части должно совпадать с числом элементов справа от равно.

А если в левой части указать лишь одну переменную, получим кортеж.

```
t = 1, 2, 3
print(f'{t=}, {type(t)}')
```

 **Важно!** Тип объектов может отличаться. Не только целые числа, как в примерах. Строки, любые коллекции. Ошибки это не вызовет. Но для повышения читаемости рекомендуется не смешивать разные типы данных при присваивании одной строкой.

- Множественное сравнение

Аналогично присваиванию можно сравнить несколько переменных внутри конструкции if.

```
a = b = c = 42
# if a == b and b == c:
if a == b == c:
    print('Полное совпадение')
```

Запись становится короче, т.к. исключается команда and внутри сравнения. Работает подобная запись не только с проверкой на равенство, но и с другими операциями.

```
if a < b < c:
    print('b больше a и меньше c')
```

Проверяем, что b больше a и b меньше c.

Плохие однострочники

А теперь несколько примеров плохого кода. Да, он будет работать, т.к. Python прощает ошибки. Но читать такой код будет сложно. И есть большая вероятность, что коллеги по проекту не простят подобный код.

```
a = 12; b = 42; c = 73
if a < b < c: b = None; print('Ужасный код')
```

Перечислили несколько операций присваивания через точку с запятой в одной строке. Такая запись противоречит PEP-8. Скорее это защита от поломки, если разработчик пришёл из другого языка и привык ставить точку с запятой.

Во второй строке не перешли на новую строку с отступами после двоеточия. Так же идёт против PEP-8. Кроме того подобные условия, циклы или функции не смогут содержать более одной вложенной строки. Если конечно не начать добавлять точки с запятой.



Очень важно! Отсутствие перехода на новую строку после двоеточия и запись нескольких строк кода в одну через точку с запятой — плохой стиль программирования. Будьте готовы получить “неудовлетворительно” за подобные антипаттерны во время учёбы и отказ в трудоустройстве во время собеседования!

Задание

Перед вами несколько строк кода. Напишите что по вашему мнению выведет print, не запуская код. У вас 3 минуты.

```
data = {10, 9, 8, 1, 6, 3}
a, b, c, *d, e = data
print(a, b, c, d, e)
```


2. Итераторы

С итераторами мы уже знакомы. Любая Python коллекция будь то список, словарь, строка и т.п. предоставляют интерфейс итератора. Если коллекцию можно передать в цикл `for in` для последовательного перебора элементов, значит коллекция итерируемая, поддерживает интерфейс итерации. При этом у каждой коллекции может быть свой интерфейс. Списки, строки, кортежи возвращают элементы слева направо, от нулевого индекса к последнему. Множества возвращают элементы в случайном порядке.



Секрет! На самом деле порядок вывода элементов множества не случаен. Он зависит от результата работы встроенного хэша. Хэш функция определяет в какое место множества будет помещён элемент и возвращает их в порядке возрастания хэша.

Просто этот порядок может не совпадать со значением элементов.

Что касается словарей, они поддерживают сразу три интерфейса итерации: по ключам, по значениям и по парам ключ-значение. Вспомните методы `keys`, `values` и `items`.

Функции `iter()` и `next()`

В Python объект является итерируемым, если поддерживает работу дандер методов `__iter__` (или `__getitem__`) и `__next__`. Первый метод должен возвращать объект итератор. Второй, `next` — возвращает очередной элемент коллекции или исключение `StopIteration`. Рассмотрим подробнее.

- Функция `iter`

Функция `iter` имеет формат `iter(object[, sentinel])`. `object` является обязательным аргументом. Если объект не реализует интерфейс итерации через методы `__iter__` или `__getitem__`, получим ошибку `TypeError`.


```
a = 42
iter(a)  # TypeError: 'int' object is not iterable
```

Получим итератор списка

```
data = [2, 4, 6, 8]
list_iter = iter(data)
print(list_iter)
```

Напрямую извлечь данные из итератора не получится. Python сообщает, что переменная `list_iter` указывает на `<list_iterator object at 0x0000025383D29400>`, т.е. объект итератор списка. Для кортежа функция `iter` вернёт `tuple_iterator`, для множеств `set_iterator`, а например для `dict.items()` вернётся `dict_itemiterator`. Один из простейших способов получить элементы - распаковать итератор через звёздочку.

```
data = [2, 4, 6, 8]
list_iter = iter(data)
print(*list_iter)
print(*list_iter)
```

 **Внимание!** Обратите внимание, что итератор является одноразовым объектом. Получив все элементы коллекции один раз он перестает работать. Для повторного извлечения элементов необходимо создать новый итератор.

Второй параметр функции `iter` — `sentinel` передают для вызываемых объектов-итераторов. Параметр указывает в какой момент должна быть завершена итерация, при совпадении возвращаемого значения со значением `sentinel`.

```
data = [2, 4, 6, 8]
list_iter = iter(data, 6)  # TypeError: iter(v, w): v must be callable
```

Список не является функцией, его нельзя вызвать. Получили ошибку `TypeError`. Один из вариантов работы функции `iter` с двумя параметрами — чтение бинарного файла блоками фиксированного размера до тех пор, пока не будет достигнут конец файла.

```
import functools

f = open('mydata.bin', 'rb')
for block in iter(functools.partial(f.read, 16), b''):
    print(block)
f.close()
```

Подробнее работу с файлами разберём на одной из следующих лекций. А сейчас коротко. Открыли бинарный файл на чтение байт. В цикле считываем блоки по 16 байт и выводим их на печать. Чтение прекратится после считывание пустого байта b'', окончания файла. После этого выходим из цикла и закрываем файл.

- Функция next

Функция next имеет формат next(iterator[, default]). На вход функция принимает итератор, который вернула функция iter. Каждый вызов функции возвращает очередной элемент итератора.

```
data = [2, 4, 6, 8]
list_iter = iter(data)
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
print(next(list_iter))
print(next(list_iter)) # StopIteration
```

При завершении элементов выбрасывается исключение StopIteration. Данное исключение служит указанием циклу for in о завершении работы. Каждый раз когда происходит перебор коллекции в цикле создаётся исключение. Но цикл обрабатывает его как сигнал для завершения итерации и перехода к следующему блоку кода. Исключение не останавливает программу.

Второй параметр функции next нужен для возврата значения по умолчанию вместо выброса исключения StopIteration

```
data = [2, 4, 6, 8]
list_iter = iter(data)
print(next(list_iter, 42))
print(next(list_iter, 42))
print(next(list_iter, 42))
print(next(list_iter, 42))
print(next(list_iter, 42))
print(next(list_iter, 42))
```

Промежуточный итог. Любая коллекция в Python поддерживает интерфейс итерации, т.е. перебор элементов в цикле for in. Для этого она возвращает итератор, который последовательно возвращает следующий, next элемент. Отдельно управлять итерацией позволяют функции iter и next. Кроме того при ООП можно создавать свои классы, поддерживающие итерации. Подробнее об этом в рамках соответствующей лекции.

Задание

Перед вами несколько строк кода. Напишите, что выведет каждая из строк, не запуская код. У вас 3 минуты.

```
data = {"один": 1, "два": 2, "три": 3}
x = iter(data.items())
print(x)
y = next(x)
print(y)
z = next(iter(y))
print(z)
```

3. Генераторы

Возвращаемся к однострочникам и рассмотрим генераторы в Python.



Важно! Генератор не обяз быть однострочником.

Генераторы похожи на итераторы тем, что возвращают некую последовательность значений. Отличие в том, что итераторы чаще всего возвращают коллекции. Т.е. коллекция хранит все данные и тратит на хранение память. Далее коллекция возвращает хранимые элементы через интерфейс итерации. Генераторы не хранят данные в памяти, а вычисляют их по мере необходимости, чтобы вернуть очередное значение.

С одним из генераторов мы уже знакомы. Это объект, возвращаемый функцией `range`. При создании генератора мы указываем диапазон перебираемых целых чисел, но не сохраняем их в памяти. Каждое из значений генерируется на очередном витке цикла.

```
a = range(0, 10, 2)
print(f'{a=}, {type(a)=}, {a.__sizeof__()=}, {len(a)}')
b = range(-1_000_000, 1_000_000, 2)
print(f'{b=}, {type(b)=}, {b.__sizeof__()=}, {len(b)}')
```

Генератор `a` на пять значений и генератор `b` на 1 млн. значений занимают одинаковое место в памяти.

Генераторные выражения

Генераторные выражения Python позволяют создать собственный генератор, перебирающий значения.

```
my_gen = (chr(i) for i in range(97, 123))
print(my_gen)          # <generator object <genexpr> at
                        # 0x000001ED58DD7D60>
for char in my_gen:
    print(char)
```

Для создания генераторного выражения используют круглые скобки, внутри которых прописывается логика выражения. В нашем примере циклический перебор целых чисел от 97 до 122 и возврат символов из таблицы ASCII с соответствующими кодами.

Комбинации for и if в генераторах и выражениях

В общем виде генератор можно записать как:

```
gen = (expression for expr in sequence1 if condition1
        for expr in sequence2 if condition2
        for expr in sequence3 if condition3
        ...
        for expr in sequenceN if conditionN)
```

Если расписать выражение в обычном коде, получим следующий код:

```
for expr in sequence1:
    if not condition1:
        continue
    for expr in sequence2:
        if not condition2:
            continue
        ...
        for expr in sequenceN:
            if not conditionN:
                continue
```

Каждый следующий цикл вложен в предыдущий. Логическая проверка определяет добавлять элемент в вывод или опустить его — `continue`. При этом логические проверки могут отсутствовать для любых циклов в любом порядке.

При работе с генераторами стоит помнить, что для каждого витка внешнего цикла внутренний перебирает все элементы от начала до конца. Например:

```
x = [1, 1, 2, 3, 5, 8, 13]
y = [1, 2, 6, 24, 120, 720]
print(f'{len(x)=}\t{len(y)=}')
mult = (i + j for i in x if i % 2 != 0 for j in y if j != 1)
res = list(mult)
print(f'{len(res)=}\n{res}')
```

Создаём генератор на основе двух списков `x` и `y`. 7 элементов в первом списке и 6 во втором. Всего 13. Генератор считает сумму пар элементов.

Генератор перебирает все значения списка `x` и оставляет только нечётные. Исключаем 2 и 8, т.е. оставляем 5 из 7 элементов списка для вычисления суммы. В списке `y` исключаем единицу, т.е. оставляем 5 из 6 элементов. Новичок может подумать, что на выходе получим 10 элементов - 5 из `x` и 5 из `y`. Но циклы вложены друг в друга, следовательно количество элементов на выходе $5 \times 5 = 25$. А асимптотика данного генератора квадратичная, $O(N \times M)$, где `N` и `M` - длина списков `x` и `y`.



Важно! На асимптотическую сложность генератора влияют только количество циклов. Наличие `if` проверок конечно же замедляет генерацию значений. Но `if` воспринимается как константа в вычислении асимптотики. 4 вложенных цикла без проверок будут иметь асимптотику 4 степени, а 3 цикла с 3 проверками — асимптотику 3-й степени. Не стоит злоупотреблять количеством вложенных циклов.

Допустимые размеры односточника

РЕР-8! Ограничение в 80 (120) символов на строку касается и генераторов. Если ваш код выходит за границу, попробуйте его упростить. Если упрощение невозможно, стоит перейти от однострочного генераторного выражения к функции генератору. Их мы рассмотрим далее на уроке.

Аналогичные ограничения касаются и рассматриваемых далее генераторов списков, множеств и словарей.

List comprehensions

Что будет, если генераторное выражение записать не в круглых скобках, а в квадратных? Получим list comprehensions. Другие названия: list comp, генератор списков, списковое включение. И нет, это не генераторное выражение. Генератор списков полностью формирует список с элементами до его присваивания переменной слева от знака равно.

```
my_listcomp = [chr(i) for i in range(97, 123)]
print(my_listcomp)  # ['a', 'b', 'c', 'd', ..., z]
for char in my_listcomp:
    print(char)
```

Как и генераторные выражения списковые включения поддерживают несколько циклов и логические проверки для каждого из циклов. Можно воспринимать их как синтаксический сахар, более короткую запись. Например выбираем все чётные числа из исходного списка и складываем их в результирующий.

Длинный код:

```
data = [2, 5, 1, 42, 65, 76, 24, 77]
res = []
for item in data:
    if item % 2 == 0:
        res.append(item)
print(f'{res = }')
```

Аналогичное решение, но с использованием синтаксического сахара listcomp:

```
data = [2, 5, 1, 42, 65, 76, 24, 77]
res = [item for item in data if item % 2 == 0]
print(f'{res = }')
```

1. Не создаём пустой список в начале.
2. Не пишем двоеточия после цикла и логической проверки.
3. Исключаем метод append.

Итого вместо 4 строк кода получаем одну.

Генераторные выражения или генерация списка

В примере из раздела “генераторные выражения” мы обернули генератор функцией `list`, чтобы сохранить значения в список. Можно воспринимать это как антипример кода. Какой же пример является верным? Если на выходе нужен готовый список, оптимальным будет следующий код:

```
x = [1, 1, 2, 3, 5, 8, 13]
y = [1, 2, 6, 24, 120, 720]
print(f'{len(x)=}\t{len(y)=}')
res = [i + j for i in x if i % 2 != 0 for j in y if j != 1]
print(f'{len(res)=}\n{res}')
```

А если нам не нужны все элементы разом. Например мы в дальнейшем хотим перебирать значения по одному в цикле. В этом случае подойдет генераторное выражение без преобразования в список.

```
x = [1, 1, 2, 3, 5, 8, 13]
y = [1, 2, 6, 24, 120, 720]
print(f'{len(x)=}\t{len(y)=}')
mult = (i + j for i in x if i % 2 != 0 for j in y if j != 1)
for item in mult:
    print(f'{item = }')
```



Важно! При написании кода заранее решите нужна вам сгенерированная коллекция целиком или нет. Не стоит тратить память на хранение всех элементов, если вы ими не пользуетесь одновременно.

Set comprehensions

Кроме синтаксического сахара для генерации списков можно создавать множества в одну строку. Синтаксис аналогичен примерам выше. Изменяются лишь скобки. Для множеств используются фигурные.

```
my_setcomp = {chr(i) for i in range(97, 123)}
print(my_setcomp)  # {'f', 'g', 'b', 'j', 'e', ... }
for char in my_setcomp:
```



```
print(char)
```

Мы также перебираем элементы в цикле. Также можно использовать вложенные циклы. Также для каждого цикла может быть проверка на включение элемента в множество.

Стоит обратить внимание на следующие особенности:

- порядок элементов внутри множества может не совпадать с порядком добавления элементов.
- множество хранит только уникальные значения

```
x = [1, 1, 2, 3, 5, 8, 13]
y = [1, 2, 6, 24, 120, 720]
print(f'{len(x)=}\t{len(y)=}')
res = {i + j for i in x if i % 2 != 0 for j in y if j != 1}
print(f'{len(res)=}\n{res}')
```

Как и в примерах выше для генерации множества перебрали все возможные комбинации пар *x* и *y* списков. Но в итоге осталось не 25 элементов, а 19 уникальных. 6 дублирующих элементов не были добавлены в множество, но время на их обработку было затрачено. Асимптотика не улучшилась.

Dict comprehensions

Ещё один вариант синтаксического сахара — генерация словаря.

```
my_dictcomp = {i: chr(i) for i in range(97, 123)}
print(my_dictcomp)  # {97: 'a', 98: 'b', 99: 'c', ... }
for number, char in my_dictcomp.items():
    print(f'dict[{number}] = {char}')
```

Запись похожа на создание множества, но в качестве выражения для добавления указываются две переменные через двоеточие: *key*: *value*. Благодаря такой записи Python понимает, что надо создать словарь.



Важно! Стоит помнить, что ключи словаря должны быть объектами неизменяемого типа.

Во всём остальном для генерации словарей в одну строку действуют те же правила, что и для других типов данных.

Задание

Перед вами несколько строк кода. Напишите что по вашему мнению выведет print, не запуская код. У вас 3 минуты.

```
data = {2, 4, 4, 6, 8, 10, 12}
res1 = {None: item for item in data if item > 4}
res2 = (item for item in data if item > 4)
res3 = [[item] for item in data if item > 4]
print(res1, res2, res3)
```

4. Создание функции генератора

Рассмотрим создание генератора не в одну строку, а как отдельную функцию. Например нам надо посчитать факториал чисел от одного до n.

Прежде чем создавать генератор, создадим обычную функцию, которая вернёт список чисел.

```
def factorial(n):
    number = 1
    result = []
    for i in range(1, n + 1):
        number *= i
        result.append(number)
    return result

for i, num in enumerate(factorial(10), start=1):
    print(f'{i}! = {num}')
```

Внутри функции создали переменную для хранения очередного числа и результирующий список. Далее в цикле перебираем числа от одного до n включительно. Число number умножается на очередное число, вычисляется следующий по порядку факториал. Результат помещается в список. По завершении

цикла возвращаем список ответов.

Получив нужное количество значений в цикле выводим факториалы и их значения. Код отлично работает, но есть но. Мы не используем все факториалы сразу, а последовательно выводим их на печать. Если бы у нас был однострочный listcomp, достаточно было бы поменять квадратные скобки на круглые и получить генераторное выражение. В нашем примере также заменим функцию на генератор.

Команда yield

Как вы помните команда return возвращает готовый результат работы функции и завершает её работу. Зарезервированное слово yield превращает функцию в генератор. Значение после yield возвращается из функции. Сама функция запоминает своё состояние: строку, на которой остановилось выполнение, значения локальных переменных. Повторный вызов функции продолжает работу с момента остановки.

Изменим функцию для получения факториала чисел, превратив её в генератор.

```
def factorial(n):
    number = 1
    for i in range(1, n + 1):
        number *= i
        yield number

for i, num in enumerate(factorial(10), start=1):
    print(f'{i}! = {num}')
```

Теперь внутри функции не создаётся пустой список для результатов. В цикле вычисляется факториал очередного числа. Далее команда yield возвращает значение. Следующий вызов вернёт функцию к циклу for для вычисления очередного числа.

Как вы помните, если в функции отсутствует команда return Python в конце тела функции добавляет return None. Явная или неявная, как в нашем примере, команда return завершает работу генератора вызовом исключения StopIteration.

Функции `iter` и `next` для генераторов

Уже знакомые по сегодняшнему уроку функции `iter` и `next` могут работать с созданными генераторами. Например так:

```
my_iter = iter(factorial(4))
print(my_iter)
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
print(next(my_iter))
```

Задание

Перед вами несколько строк кода. Напишите что по вашему мнению выведет `print`, не запуская код. У вас 3 минуты.

```
def gen(a: int, b: int) -> str:
    if a > b:
        a, b = b, a
    for i in range(a, b + 1):
        yield str(i)

for item in gen(10, 1):
    print(f'{item} = ')
```

Вывод

На этой лекции мы:

1. Разобрали решения задач в одну строку
2. Изучили итераторы и особенности их работы

3. Узнали о генераторных выражениях и генераторах списков, словарей, множеств
4. Разобрали создание собственных функций генераторов.

Краткий анонс следующей лекции

- 1.

Домашнее задание

Возьмите несколько задач из прошлых уроков, где вы создавали функции и сделайте из них генераторы. Внесите правки в код, чтобы он работал без ошибок в новой реализации.



Подсказка: замените `return` на `yield` в теле функций.

Оглавление

На этой лекции мы	2
Дополнительные материалы к лекции	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции. Введение	4
1. Ещё раз про import	4
Переменная sys.path	5
Антипримеры импорта	5
Использование from и as	6
Плохой import * (импорт звёздочка)	7
Переменная __all__	9
Задание	10
2. Виды модулей	10
Встроенные модули	11
Свои модули	12
Пишем свой модуль: __name__ == '__main__'	13
Разбор плохого импорта	14
Создание пакетов и их импорт	15
Разница между модулем и пакетом	16
Варианты импорта	17
Задание	18
3. Некоторые модули “из коробки”	19
Модуль sys	19
Запуск скрипта с параметрами	19
Модуль random	21
Задание	22

На этой лекции мы

1. Разберём работу с модулями в Python
2. Изучим особенности импорта объектов в проект
3. Узнаем о встроенных модулях и возможностях по созданию своих модулей и пакетов
4. Разберём модуль random отвечающий за генерацию случайных чисел

Дополнительные материалы к лекции

Стандартная библиотека Python. Документация.

<https://docs.python.org/3/library/index.html>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрали решения задач в одну строку
2. Изучили итераторы и особенности их работы
3. Узнали о генераторных выражениях и генераторах списков, словарей, множеств
4. Разобрали создание собственных функций генераторов.

Термины лекции

- **Модуль** — это файл, содержащий определения и операторы Python. Во время исполнения же модуль представлен соответствующим объектом, атрибутами

которого являются объявления, присутствующие в файле и объекты, импортированные в этот модуль откуда-либо.

- **Пакет** — это набор взаимосвязанных модулей (при этом стоит уточнить, что сам пакет тоже является модулем), предназначенных для решения задач определенного класса некоторой предметной области. Пакеты — это способ структурирования пространства имен модулей Python с помощью «точечных имен модулей». Пакет представляет собой папку, в которой содержатся модули и другие пакеты и обязательный файл `__init__.py`, отвечающий за инициализацию пакета.

Подробный текст лекции

Введение

Прежде чем начать, немного о названиях. Когда вы пишете код на Python и сохраняете его в файл с расширением `py`, получаем программу на Python. Другие названия для программы в файле — скрипт, Python скрипт. Помимо скрипта вы можете встретить термины модуль и пакет.

Модули можно импортировать в другие файлы для использования написанного ранее кода в другом проекте. По сути любой Python скрипт является модулем. Для разработчика на Python можно утверждать, что программа, скрипт и модуль — синонимы.

Пакет представляет из себя набор модулей. В Python любой пакет одновременно является модулем. Но не каждый модуль считается пакетом. Подробнее о модулях и пакетах далее в лекции.

1. Ещё раз про `import`

Мы уже использовали зарезервированное слово `import` для добавление в собственный код расширенного функционала. Разберём `import` подробнее.



PEP-8! Строки импорта рекомендуется писать в самом начале файла, оставляя 1-2 пустые строки после него.


```
import sys

print(sys)
print(sys.builtin_module_names)
print(*sys.path, sep='\n')
```

Мы импортировали модуль `sys` из стандартной библиотеки Python. Механизм импорта выполняет поиск указанного имени (в нашем примере — “`sys`”) в нескольких местах. В первую очередь проверяется список встроенных модулей, который хранится в `sys.builtin_module_names`. Если имя модуля не найдено, проверка ведётся в каталогах файловой системы, которые перечислены в `sys.path`.



Важно! Обычно (но не всегда) имя модуля заканчивается расширением `.py`. При импорте расширение не указывается.

В результате импорта имя `sys` было добавлено в текущую, глобальную область видимости. Для того, чтобы получить доступ к переменным, функциям, классам и т.п. содержимому модуля используется точечная нотация: `имя_модуля<точка>имя_объекта`.

Обратиться к объекту внутри модуля напрямую при “обычном” импорте не получится. Подобный подход защищает от конфликта имён. Если и в вашем файле и в импортируемом модуле есть функция `func()`, конфликта имён не будет. Свою функцию вызываете как `func()`, а функцию из модуля как `module_name.func()`

- **Переменная `sys.path`**

Содержимое переменной `sys.path` формируется динамически. В качестве первого адреса указывается путь до основного файла. Например если вы используете Unix подобную ОС и ваш скрипт расположен по адресу `/home/user/project`, именно этот путь будет первым в списке поиска модулей.

Далее в `sys.path` перечислены пути из `PYTHONPATH` и пути, указанные при установке Python и создании виртуального окружения. Таким образом Python ищет импортируемый модуль практически во всех местах, где этот модуль мог быть установлен.



Важно! Если создать собственный файл с именем аналогичным имени модуля, Python импортирует ваш файл, а не модуль. Строго не рекомендуется использовать для своих файлов имена встроенных модулей. В редких исключительных ситуациях стоит добавлять символ подчёркивания в конце имени, чтобы избежать двойного именования.

- **Антипримеры импорта**

Взгляните на антипример. Мы создали файл `random.py` со следующим кодом.

```
def randint(*args):  
    return 'Не то, что вы искали!'
```

Импортируем модуль `random` в основном файле проекта и попробуем сгенерировать случайное число от 1 до 6.

```
import random  
  
print(random.randint(1, 6))
```

В результате работы получим “Не то, что вы искали!”. Подробнее о модуле для генерации случайных чисел поговорим далее в этом уроке.



PEP-8! При импорте нескольких модулей каждый указывается с новой строки.

Правильно:

```
import sys  
import random
```

Неправильно:

```
import sys, random
```

Использование `from` и `as`

Помимо обычного импорта можно использовать более подробную форму записи. Зарезервированное слово `from` указывает на имя модуля или пакета, далее `import` и имена импортируемых объектов.

```
from sys import builtin_module_names, path  
  
print(builtin_module_names)  
print(*path, sep='\n')
```

Теперь при обращении к импортированным объектам не нужно указывать имя модуля. Мы явно добавили их в наш код, включили имена в область видимости.



PEP-8! Конструкция `from import` допускает перечисление импортируемых имён объектов через запятую в одной строке. После `from` всегда указывается один модуль.

Кроме выборочного импорта можно создавать псевдонимы для объектов через зарезервированное слово `as`. При этом доступ к объекту будет возможен только через псевдоним. Один объект — одно имя.

```
import random as rnd
from sys import builtin_module_names as bmn, path as p

print(bmn)
print(*p, sep='\n')
print(rnd.randint(1, 6))
print(path)    # NameError: name 'path' is not defined
print(sys.path) # NameError: name 'sys' is not defined
```

В первой строке импортировали модуль `random` и присвоили ему имя `rnd` внутри текущей области видимости. Во второй строке импортировали переменную `builtin_module_names` под именем `bmn` и переменную `path` под именем `p`. Последние две строки вызывают ошибку имени. Мы не можем обратиться к переменной `path`, потому что дали ей другое имя — `p`. И обращение к модулю `sys` не работает, ведь мы его не импортировали. Только объекты из модуля.



Важно! Не стоит давать переменным короткие понятные лишь вам имена. Код должен легко читаться другими разработчиками. Исключения — общепризнанные сокращения, например `import numpy as np`.

Плохой `import *` (импорт звёздочка)

Ещё один вариант импорта: `from имя_модуля import *`

Подобная запись импортирует из модуля все глобальные объекты за исключением тех, чьи имена начинаются с символа подчёркивания. Рассмотрим на примере.

- Файл `super_module.py`

```

from random import randint

SIZE = 100
_secret = 'qwerty'
__top_secret = '1q2w3e4r5t6y'


def func(a: int, b: int) -> str:
    z = f'В диапазоне от {a} до {b} получили {randint(a, b)}'
    return z

result = func(1, 6)

```

В модуле есть следующие объекты:

- глобальная функция randint
- глобальная константа SIZE
- глобальная защищенная переменная _secret
- глобальная приватная переменная __top_secret
- глобальная функция func
- локальные параметры функции a и b
- локальная переменная функции z
- глобальная переменная result

 **Внимание!** Если название объекта (переменной, функции и т.п.) начинается с символа подчёркивания, объект становится защищённым. Если имя начинается с двух подчёркиваний, объект становится приватным. Объекты без подчёркивания в начале имени — публичные. Подробнее разберём на лекциях по ООП.

Импортируем модуль в основной файл программы через звёздочку и попробуем выполнить несколько операций.

- **Файл main.py**

```

from super_module import *

SIZE = 49.5

print(f'{SIZE = }\n{result = }')
print(f'{z = }')    # NameError: name 'z' is not defined
print(f'{_secret = }')    # NameError: name '_secret' is not defined
print(f'{func(100, 200) = }\n{randint(10, 20) = }')

```

```
def func(a: int, b: int) -> int:
    return a + b

print(f'{func(100, 200) = }')
```

Первая строка импортирует в файл все глобальные публичные объекты. Далее мы определяем константу SIZE. В этот момент значение константы из модуля затирается новым значением. Возник конфликт имён и Python разрешил его в пользу нового значения константы. При этом содержимое переменной result берётся из модуля super_module, т.к. других определений переменной нет в файле. При попытке обратиться к локальной и защищённой переменным получаем ошибки. Они не были импортированы “звёздочкой”.

Далее мы вызываем функции func и randint. Они верно отработывают код, т.к. обе были импортированы из внешнего модуля.

В финале создаём свою функцию func. Возникает очередной конфликт имён, который Python разрешает в пользу новой функции. В результате вызов func() после её определения возвращает совсем другой результат, чем вызовом ранее.

Промежуточный итог. Использование стиля from module import * зачастую приводит к неожиданным результатам, затрудняет отладку кода и мешает верному пониманию работы программы. Использовать подобный приём стоит в редких особенных случаях. Кроме того импорт всех доступных объектов может значительно замедлить работы программы, если таких объектов очень много.

Переменная `__all__`

При необходимости разработчик модуля может явно указать какие объекты нужно импортировать при использовании стиля from module import *. Для этого используется магическая переменная `__all__` (два нижних подчёркивания до, слово all и два нижних подчёркивания после). Изменим код модуля super_module.py, добавив строку с `__all__`

Файл super_module.py

```
from random import randint
```

```

__all__ = ['func', '_secret']

SIZE = 100
_secret = 'qwerty'
_top_secret = '1q2w3e4r5t6y'

def func(a: int, b: int) -> str:
    z = f'В диапазоне от {a} до {b} получили {randint(a, b)}'
    return z

result = func(1, 6)

```

Переменной `__all__` присваивается список имён объектов, заключённых в кавычки, т.е. `str` типа. В основной модуль попадут только указанные в списке имена, независимо от того являются они публичными, защищёнными или приватными. При этом объект должен быть глобальным. Если указать в списке имя локального объекта, например переменную `z` — локальную переменную функции `func`, получим ошибку.

Список `__all__` в приведённом примере используется для формирования списка импортируемых объектов модуля. Кроме этого `__all__` применяется для импорта модулей из пакета. Рассмотрим вариант импорта модулей из пакета далее на лекции.

Задание

Перед вами пример кода. Какие переменные будут доступны после импорта для работы в основном файле? У вас три минуты.

```

import sys
from random import *
from super_module import func as f

```

2. Виды модулей

Попробуем добавить некоторую системность в модули. В Python есть:

- встроенные модули,
- установленные внешние модули,
- модули, созданные разработчиком, свои.

Кроме того каждый вид модулей может быть внутри пакета, сгруппированной коллекции модулей. Разберём подробнее.

Встроенные модули

Один из плюсов языка Python — его батарейки. Так принято называть стандартную библиотеку языка. По сути библиотека представляет обширный набор пакетов и модулей для решения широкого спектра задач.

Некоторые модули стандартной библиотеки мы уже использовали. С другими будем знакомиться в рамках последующих уроков. Финальный урок курса целиком посвятим разговору о стандартной библиотеке. А пока определимся с несколькими вещами:

1. Стандартная библиотека устанавливается вместе с интерпретатором. Дополнительные манипуляции по установке не требуются. Всё работает “из коробки”.¹
2. Для использования модуля стандартной библиотеки достаточно его импортировать в ваш код.
3. Большинство частых задач легко решаются средствами стандартной библиотеки. Достаточно обратиться к [справке](#) и найти нужный модуль.
4. Некоторые модули стандартной библиотеки разрабатывались настолько давно, что не отвечают современным требованиям решения задач. В таком случае на помощь приходят внешние решения. И наоборот. Каждое обновление Python вносит улучшения в библиотеку, зачастую более эффективные, чем внешние решения.

¹ Установщики Python для платформы Windows обычно включают в себя всю стандартную библиотеку, а также множество дополнительных компонентов. Для Unix-подобных операционных систем Python обычно предоставляется в виде набора пакетов, поэтому может потребоваться использование инструментов упаковки, поставляемых с операционной системой, для получения некоторых или всех дополнительных компонентов.



PEP-8! Импорт модулей стандартной библиотеки пишется в начале файла, до импорта внешних и своих модулей. После импорта оставляют пустую строку, даже если далее идёт импорт модулей не из библиотеки.

Свои модули

А теперь пара слов о том как получить свой модуль. В вашей любимой IDE создаём файл с расширением `py`, пишем в нём код и сохраняем результат. Готово! Вы создали свой модуль. Ранее на лекции мы создали файл `super_module.py` и импортировали его в `main.py`, т.е. работали с файлом как с модулем.

Обычно модуль должен решать одну задачу или несколько однотипных задач.

Стандартная структура любого модуля следующая:

- документация по модулю в виде многострочного комментария (три пары двойных кавычек),
- импорт необходимых пакетов, модулей, классов, функций и т.п. объектов,
- определение констант уровня модуля,
- создание классов модуля при ООП подходе,
- создание функций модуля,
- определение переменных модуля,
- покрытие тестами, если оно не вынесено в отдельный пакет,
- `main` код.

В зависимости от решаемой задачи некоторые пункты могут отсутствовать.

Учебный пример модуля ниже:

Файл `base_math.py`

```
"""Four basic mathematical operations.

Addition, subtraction, multiplication and division as functions.
"""

_START_SUM = 0
_START_MULT = 1
_BEGINNING = 0
_CONTINUATION = 1


def add(*args):
    res = _START_SUM
    for item in args:
        res += item
```



```

    return res

def sub(*args):
    res = args[_BEGINNING]
    for item in args[_CONTINUATION:]:
        res -= item
    return res

def mul(*args):
    res = _START_MULT
    for item in args:
        res *= item
    return res

def div(*args):
    res = args[_BEGINNING]
    for item in args[_CONTINUATION:]:
        res /= item
    return res

```

Как вы заметили, в файле нет классов, только функции. Так же нам не понадобились переменные уровня модуля. Тесты мы не стали писать, т.к. не добрались до темы тестирования. Осталось добавить main код.

Пишем свой модуль: `__name__ == '__main__'`

Давайте всё же проверим работоспособность кода. Сделаем по 1-2 вызова каждой функции в том же файле и выведем результат на печать. Визуально убедимся, что код работает верно.

```

print(f'{add(2, 4) = }')
print(f'{add(2, 4, 6, 8) = }')
print(f'{sub(10, 2) = }')
print(f'{mul(2, 2, 2, 2, 2) = }')
print(f'{div(-100, 5, -2) = }')

```

Визуальное тестирование прошло успешно. Импортируем модуль в основной код нашего учебного проекта, в файл main.py

```
import base_math

x = base_math.mul    # Плохой приём
y = base_math._START_MULT    # Очень плохой приём
z = base_math.sub(73, 42)
print(x(2, 3))
print(y)
print(z)
```

Запускаем файл и наблюдаем вывод “принтов” из файла base_math раньше наших расчётов в “мейне”. Дело в том, что команда import запускает импортируемый модуль. В результате лишние вызовы функций, а следовательно более медленное выполнение кода. Для решения проблемы необходимо внести правки в файл base_math.py.

```
...
if __name__ == '__main__':
    print(f'{add(2, 4) = }')
...
```

После определения функций, но перед вызовом print добавили логическую проверку. Строки с print после проверки и до конца файла сдвинули на 4 пробела, поместили внутрь блока проверки.

Магическая переменная __name__ содержит полное имя модуля. Это имя используется для уникальной идентификации модуля в системе импорта.

Когда мы запускаем сам модуль base_math, в переменную __name__ попадает имя “__main__”. Оно указывает, что файл запущен, а не импортирован. Никакой разницы в работе модуля не произошло.

Когда модуль импортируется в другой файл, в переменную __name__ попадает его имя, название файла без расширения. import base_math также выполняет код в файле. Присваиваются значения константам, инициализируются функции. Но логическая проверка if __name__ == '__main__': возвращает ложь и дальнейший код пропускается.

Отличной. Добавление “мейн” проверки позволило завершить создание модуля. Теперь запуск main.py работает так, как мы ожидаем.

Разбор плохого импорта

Вернёмся к коду в `main.py` и разберём пару антипримеров.

1. `x = base_math.mul`

Мы передали (не путайте с вызвали) в переменную `x` функцию `mul`. Теперь для умножения можно использовать более короткое имя. Но `x` не является понятным для других разработчиков именем.

Логичнее было бы написать:

```
multiplication = base_math.mul
```

Если же нужно короткое имя, то

```
mul = base_math.mul
```

Но и тут мы сталкиваемся с проблемой. В большом файле может быть трудно заметить присвоение нового имени для функции. Самым логичным вариантом будет использование сочетания `from base_math import mul`. Или `from base_math import mul as mult`, если хотим дать новое имя.

2. `y = base_math._START_MULT`

В отличие от `from module import *` обычный импорт позволяет получить доступ к защищённым переменным. Мы смогли прочитать константу `_START_MULT`. В Python действует следующее правило. Все программисты взрослые люди. Если один из них нарушает соглашения по написанию кода, значит он готов взять на себя ответственность за возможные проблемы.

Важно! Не используйте защищённые и приватные объекты за пределами модуля, в котором они созданы. Особенно если это не ваш модуль. Исключение — прописанное в модуле указание на использование.

Создание пакетов и их импорт

Продолжим нашу идею с учебным математическим модулем. Наш проект развивается и решено добавить целочисленное деление и возведение в степень. А чтобы модуль не разрастался до бесконечности, продвинутую математику будем

хранить в отдельном файле `advanced_math.py`

```
"""Two advanced mathematical operations.

Integer division and exponentiation."""

__all__ = ['div', 'exp']
_BEGINNING = 0
_CONTINUATION = 1

def div(*args):
    res = args[_BEGINNING]
    for item in args[_CONTINUATION:]:
        res //= item
    return res

def exp(*args):
    res = args[_BEGINNING]
    for item in args[_CONTINUATION:]:
        res **= item
    return res

if __name__ == '__main__':
    print(f'{div(42, 4) = }')
    print(f'{exp(2, 4, 6, 8) = }')
```

Самое время объединить два схожих модуля в один пакет. Создаём директорию `mathematical` и переносим в неё оба файла: `base_math.py` и `advanced_math.py`. Далее создаём в каталоге пустой файл `__init__.py`. Пакет `mathematical` готов.

В Python любая директория с файлом `__init__.py` автоматически становится пакетом. При этом полезный функционал содержится в других питоновских файлах, а не в “инит”.

Разница между модулем и пакетом

Пакет — директория с `__init__.py` файлом и другими `py` файлами — модулями. В Python любой пакет является одновременно и модулем. Это означает, что пакет можно импортировать в проект как и модуль. Так же это означает, что пакет может хранить в себе другие пакеты — директории с “инит” файлом. Глубина вложенности

ограничена лишь здравым смыслом.

Пример пакета `sound`, содержащего пакеты `formats`, `effects` и `filters`, содержащие различные модули из официальной документации. Обратите внимание, что в проекте четыре директории и четыре файла `__init__.py`, по одному на пакет.

```
sound/                                Top-level package
  __init__.py                          Initialize the sound package
  formats/                             Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
  effects/                             Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
  filters/                             Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...
```

Подробнее про `__init__.py`

Внутри файл `__init__.py` можно прописать код, который будет выполняться при импорте пакета. Кроме того, в файл можно добавить переменную `__all__` с именами всех модулей пакета. Например мы добавим в учебный проект следующую строку:

```
__all__ = ['base_math', 'advanced_math']
```



Важно! При добавлении или удалении модулей в пакет важно вносить изменения в список `__all__` для корректной работы импорта.

Варианты импорта

Рассмотрим особенности импорта пакетов и модулей.

```
import mathematical
```

```
x = mathematical.base_math.div(12, 5)
```

Если импортировать пакет верхнего уровня, для работы с функциями необходимо указать всю цепочку через точечную нотацию: имя пакета, имя модуля, имя объекта.

Для сокращения объема кода обычно импортируют нужные модули или объекты модуля через точечную нотацию после `import`.

```
from mathematical import base_math as bm  
from mathematical.advanced_math import exp
```

```
x = bm.div(12, 5)
```

```
z = exp(2, 3)
```

В первом случае импортировали модуль `base_math` под именем `bm`. Во-втором — функцию `exp` из модуля `advanced_math` пакета `mathematical`.

Отдельно стоит упомянуть особенности импорта внутри подпакетов. Например импорт модуля `other_module` в другой модуль того же пакета можно осуществить через относительный импорт:

```
from . import other_module
```

А если модулю надо выйти из своего пакета в пакет верхнего уровня, используют вторую точку:

```
from .. import other_module
```

Или так

```
from ..other_package import other_module
```



Важно! Модуль, который является основным в вашем проекте должен использовать только абсолютные имена пакетов и модулей. Связано это с тем, что у запускаемого модуля в переменной `__name__` хранится значение `“__main__”`, а не имя модуля.

Задание

Перед вами список файлов в директории проекта. Напишите в чат какие файлы надо удалить или переименовать, а какие верные. У вас три минуты.

```
__all__.py  
__init__.py  
__main__.py  
init.py  
math.py  
random.py
```

3. Некоторые модули “из коробки”

Рассмотрим несколько модулей из стандартной библиотеки Python. Точнее даже не модули целиком, а некоторые функции из модулей.

Модуль sys

С модулем `sys` мы уже познакомились. Он относится к группе служебных в Python. Модуль `sys` обеспечивает доступ к некоторым переменным, используемым или поддерживаемым интерпретатором, а также к функциям, тесно взаимодействующим с интерпретатором. Изучать все переменные и функции не имеет смысла. Даже опытные разработчики не пользуются всеми, а их в модуле около сотни. При необходимости вы всегда сможете найти описание той или иной функции или переменной в официальной документации. Рассмотрим лишь одну переменную модуля — `argv`.

Запуск скрипта с параметрами

Python позволяет запускать скрипты с параметрами. Для этого после имени исполняемого файла указываются ключи и/или значения через пробел.

Например создадим файл `script.py` со следующим кодом.

```
print('start')  
  
print('stop')
```

Открываем консоль операционной системы и вводим команду на запуск.

```
python3 script.py
```



Важно! Для UNIX ОС используем команду `python3`. В Windows — `python`.



Важно! Не перепутайте консоль ОС и терминал интерпретатора Python. Терминал выдаёт в начале строки приветствие на ввод — тройную стрелку `>>>`.

Скрипт вывел текст в консоль и завершил работу. Научим его принимать значения из командной строки.

```
from sys import argv  
  
print('start')  
print(argv)  
print('stop')
```

Переменная `argv` содержит список. В нулевой ячейке имя запускаемого скрипта. В последующих ячейках переданные значения.

Например при запуске следующей строки:

```
python script.py -d 42 -s "Hello world!" -k 100
```

получим следующий список:

```
['script.py', '-d', '42', '-s', 'Hello world!', '-k', '100']
```

Обратите внимание, что все значения переданы как строки даже числовые. Строка “Hello world!” передана как один объект, потому что при вызове скрипта была заключена в двойные кавычки.

Переменная `argv` позволяет решать простые задачи работы с командной строкой. Если ваш будущий проект будет требовать более сложной обработки переданных в скрипт параметров, обратите внимание на встроенный в Python модуль `argparse`.

Модуль random

Модуль используется для генерации псевдослучайных чисел. Почти все функции модуля зависят от работы функции `random()`, которая генерирует псевдослучайные числа в диапазоне от нуля включительно до единицы исключительно — $[0, 1)$.



Важно! Генераторы псевдослучайных чисел модуля не должны использоваться в целях безопасности. Для обеспечения безопасности или криптографии необходимо использовать модуль `secrets`.

Для управления состоянием используют следующие 3 функции:

- `seed(a=None, version=2)` — инициализирует генератор. Если значение `a` не указано, для инициализации используется текущее время ПК. Версия 2 используется со времён Python 3.2 как основная. Не стоит менять её.
- `getstate()` — возвращает объект с текущим состоянием генератора.
- `setstate(state)` — устанавливает новое состояние генератора, принимая на вход объект, возвращаемый функцией `getstate`.

Рассмотрим несколько часто используемых функции генерации чисел.

- `randint(a, b)` — генерация случайного целого числа в диапазоне от `a` включительно до `b` включительно — $[a, b]$.
- `uniform(a, b)` — генерация случайного вещественного числа в диапазоне от `a` до `b`. Правая граница может как входить, так и не входить в возвращаемый диапазон. Зависит от способа округления.
- `choice(seq)` — возвращает случайный элемент из непустой последовательности.
- `randrange(stop)` или `randrange(start, stop[, step])` работает как вложение функции `range` в функцию `choice`, т.е. `choice(range(start, stop, step))`. Возвращает случайное число от `start` до `stop` с шагом `step`.
- `shuffle(x)` — перемешивает случайным образом изменяемую последовательность `in place`, т.е. не создавая новую.
- `sample(population, k, *, counts=None)` — выбирает `k` уникальных элементов из последовательности `population` и возвращает их в новой последовательности. Параметр `counts` позволяет указать количество повторов элемента.

Все описанные функции представлены в листинге ниже.

```
import random as rnd
```

```

START = -100
STOP = 1_000
STEP = 10
data = [2, 4, 6, 8, 42, 73]

print(rnd.random())
rnd.seed(42)
state = rnd.getstate()
print(rnd.random())
rnd.setstate(state)
print(rnd.random())

print(rnd.randint(START, STOP))
print(rnd.uniform(START, STOP))
print(rnd.choice(data))
print(rnd.randrange(START, STOP, STEP))

print(data)
rnd.shuffle(data)
print(data)

print(rnd.sample(data, 2))
print(rnd.sample(data, 2, counts=[1, 1, 1, 1, 1, 100]))

```

Задание

Перед вами пример кода. Что выведет программа после запуска? У вас три минуты.

```

import random
from sys import argv

print(random.uniform(int(argv[1]), int(argv[2])))
print(random.randrange(int(argv[1]), int(argv[2]), int(argv[1])))
print(random.sample(range(int(argv[1]),
                           int(argv[2]),
                           int(argv[1])), 10))

```

Скрипт запущен командой: `python3 main.py 10 1010`

Вывод

На этой лекции мы:

1. Разобрали работу с модулями в Python
2. Изучили особенности импорта объектов в проект
3. Узнали о встроенных модулях и возможностях по созданию своих модулей и пакетов
4. Разобрали модуль random отвечающий за генерацию случайных чисел

Краткий анонс следующей лекции

1. Разберёмся в особенностях работы с файлами и каталогами в Python
2. Изучим функцию open для работы с содержимым файла
3. Узнаем о возможностях стандартной библиотеки для работы с файлами и каталогами

Домашнее задание

Соберите файлы прошлых уроков в отдельные директории и сделайте их пакетами. Измените сами файлы, чтобы они были импортируемыми модулями.



Подсказка. Используйте файлы `__init__.py` и проверку переменной `__name__`.

Оглавление

На этой лекции мы	3
Дополнительные материалы к лекции	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	4
Подробный текст лекции	
1. Файлы	4
Функция open()	4
Режимы работы с файлами	6
Метод close()	6
Прочие необязательные параметры функции open	7
Менеджер контекста with open	8
Чтение файла: целиком, через генератор	9
Чтение в список	9
Чтение методом read	9
Чтение методом readline	10
Чтение циклом for	11
Запись и добавление в файл	11
Запись методом write	11
Запись методом writelines	12
print в файл	13
Методы перемещения в файле	14
Метод tell	14
Метод seek	14
Метод truncate	15
Задание	16

2. Файловая система	16
Работа с каталогами	17
Текущий каталог	17
Создание каталогов	17
Удаление каталогов	18
Формирование пути	18
Чтение данных о каталогах	19
Проверка на директорию, файл и ссылку	19
Обход папок через <code>os.walk()</code>	20
Работа с файлами	21
Переименование файлов	21
Перемещение файлов	21
Копирование файлов	22
Удаление файлов	22
Задание	23
Вывод	23

На этой лекции мы

1. Разберёмся в особенностях работы с файлами и каталогами в Python
2. Изучим функцию `open` для работы с содержимым файла
3. Узнаем о возможностях стандартной библиотеки для работы с файлами и каталогами

Дополнительные материалы к лекции

Стандартная библиотека Python. Документация.

<https://docs.python.org/3/library/index.html>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрали работу с модулями в Python
2. Изучили особенности импорта объектов в проект
3. Узнали о встроенных модулях и возможностях по созданию своих модулей и пакетов
4. Разобрали модуль random отвечающий за генерацию случайных чисел

Термины лекции

- **Файл (англ. file)** — именованная область данных на носителе информации, используемая как базовый объект взаимодействия с данными в операционных системах.
- **Файловая система (англ. file system)** — порядок, определяющий способ организации, хранения и именования данных на носителях информации.

Подробный текст лекции

1. Файлы

Файл — это область данных на носителе информации (диске, флешке и т.п.). Файл используется как базовый объект взаимодействия с данными. Операционная система обращается к файлу по имени.

Функция open()

В Python для получения доступа файлу используют функцию open().

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

Функция open имеет один обязательный параметр — file. Обычно file — это объект типа str или bytes, который представляет абсолютный или относительный путь к файлу. Относительный путь считается от каталога запуска программы. Функция возвращает файловый объект или вызывает ошибку OSError, если не получилось вернуть объект.

Рассмотрим пример.

```
f = open('text_data.txt')
print(f)
print(list(f))
```

На выходе получим примерно такой вывод из переменной f:

```
<_io.TextIOWrapper name='text_data.txt' mode='r' encoding='cp1251'>
```

Файловый объект для операций текстового ввода выводу файла text_data.txt в режиме чтения в кодировке cp1251.

Для вывода информации воспользовались превращением файлового объекта в список: list(f). Подобный приём удобен для получения всего текстового файла целиком в виде элементов списка. Каждая строка лежит в новой ячейке. Но если файл имеет большие размеры, его сохранение в список может занять много времени. Кроме того будет потрачена оперативная память. Учитывайте это при открытии файлов больше, чем свободная ОЗУ.

Если не указывать режим открытия, файл открывается для чтения текста. При этом в качестве кодировки по умолчанию используется кодировка ОС. И если с режимом чтения понятно, то с кодировкой могут быть проблемы. Вот так выглядит “Привет, мир!”, сохранённый в utf-8, но открытый в cp1251 - РцСТЈРёРІРµС,, РјРёСТЈ!

Чтобы избежать проблем при работе с файлами рекомендуется при открытии указывать как минимум три параметра: название файла, режим и кодировку.

```
f = open('text_data.txt', 'r', encoding='utf-8')
print(f)
print(list(f))
```



Важно! Кодировка UTF-8 является современным стандартом для хранения и передачи текстовой информации. Если вы явно не планируете

работать с другой кодировкой, всегда указывайте `encoding='utf-8'` при открытии текстовых файлов. Это обеспечит переносимость вашего кода между различными платформами.

- **Режимы работы с файлами**

Рассмотрим доступные режимы работы с файлами.

- `'r'` — открыть для чтения (по умолчанию)
- `'w'` — открыть для записи, предварительно очистив файл
- `'x'` — открыть для эксклюзивного создания. Вернёт ошибку, если файл уже существует
- `'a'` — открыть для записи в конец файла, если он существует
- `'b'` — двоичный режим
- `'t'` — текстовый режим (по умолчанию)
- `'+'` — открыты для обновления (чтение и запись)

По умолчанию используется режим чтения текста — `"rt"`. Если не указать режим `"t"`, он автоматически добавляется к `r`, `w`, `x` или `a`. Для чтения файла побайтно указывать режим `"b"` обязательно. Режимы `'w+'` и `'w+b'` открывают файл для чтения и записи и удаляют старое содержимое. Режимы `'r+'` и `'r+b'` открывают на чтение и запись без удаления старой информации.



Важно! При открытии файла в режиме `"b"` информация предаётся в виде байт. Указывать кодировку `encoding` при этом режиме не нужно, т.к. данные не декодируются.

- **Метод `close()`**

После завершения работы с файлом необходимо освободить ресурсы. Для этого вызывается метод `close()`.

```
f = open('text_data.txt', 'a', encoding='utf-8')
f.write('Окончание файла\n')
f.close()
```

Заккрытие файла гарантирует сохранение информации на носителе.



Важно! Если в коде отсутствует метод `close()`, то даже при успешном завершении программы не гарантируется сохранение всех данных в файле.

После закрытия файла происходит высвобождение ресурсов и переменная файловый объект (в наших примерах — `f`) становится недоступна для манипуляций с файлом.

- **Прочие необязательные параметры функции `open`**

`buffering` — определяет режим буферизации. При работе с бинарными файлами можно передать `0` — отключить буферизацию. В текстовом режиме можно передать `1` — использовать буферизацию строк. Число больше единицы определяет размер буфера в байтах для двоичных файлов. По умолчанию размер буфера подстраивается под файловую систему и обычно равен 4096 или 8192 байта.

```
f = open('bin_data', 'wb', buffering=64)
f.write(b'X' * 1200)
f.close()
```

`errors` — используется только в текстовом режиме и определяет поведение в случае ошибок кодирования или декодирования. Рассмотрим несколько возможных вариантов параметра:

- `'strict'` — вызывает исключение `ValueError` в случае ошибки. Работает как значение по умолчанию.
- `'ignore'` — игнорирует ошибки кодирования. При этом игнорирование ошибок может привести к потере данных.
- `'replace'` — вставляет маркер замены (например, `'?'`) там, где есть некодировемые данные.
- `'namereplace'` — при записи заменяет неподдерживаемые символы последовательностями `\N{...}`.

```
f = open('data.txt', 'wb')
f.write('Привет, '.encode('utf-8') + 'мир!'.encode('cp1251'))
f.close()

f = open('data.txt', 'r', encoding='utf-8')
print(list(f)) # UnicodeDecodeError: 'utf-8' codec can't decode
byte 0xec in position 14: invalid continuation byte
f.close()

f = open('data.txt', 'r', encoding='utf-8', errors='replace')
print(list(f))
f.close()
```

`newline` — отвечает за преобразование окончания строки

`closefd` — указывает оставлять ли файловый дескриптор открытым при закрытии файла

`opener` — позволяет передать пользовательскую функцию для открытия файла.

Подробнее об этих параметрах можно прочитать в официальной документации, если возникнет необходимость.

Менеджер контекста `with open`

Если после открытия файла в коде возникнет ошибка, строка `f.close()` не будет выполнена. В результате ресурсы не освободятся, возможно возникнут проблемы в работе с открываемым файлом. Чтобы избежать подобных ошибок используют менеджер контекста `with`.

```
with open('text_data.txt', 'r+', encoding='utf-8') as f:
    print(list(f))
print(f.write('Пока'))      # ValueError: I/O operation on closed
                             file.
```

В отличие от прошлых примеров с `open` вместо присвоения “`f =`” в начале строки используем “`as f:`” в конце строки. Вложенный в `with` блок кода позволяет работать с файлом через файловый дескриптор `f`. При завершении вложенного блока происходит автоматическое закрытие файла. При этом даже в случае ошибки внутри блока сначала будет закрыт файл, а только потом произведено аварийное завершение программы.

Обычно вариант написания кода с менеджером контекста `with` предпочтительнее, чем напрямую с `open`. Но если вам надо одновременно работать с несколькими файлами, вариант с `open` позволит избежать нескольких уровней вложенности. В этом случае не стоит забывать про закрытие всех файлов.

Однако менеджер контекста позволяет одновременно открыть несколько файлов следующим образом:

```
with open('text_data.txt', 'r+', encoding='utf-8') as f1, \
    open('bin_data', 'rb') as f2, \
    open('data.txt', 'r', encoding='utf-8',
errors='backslashreplace') as f3:
    print(list(f1))
```

```
print(list(f2))
print(list(f3))
```

Обратите внимание на запятые между `open`. И при переносе кода на новую строку обязательно указывать обратную косую черту (бэкслеш). Так Python поймёт, что это одна длинная строка с переносами, а не несколько отдельных.

Начиная с Python 3.10 менеджер контекста поддерживает круглые скобки для группировки нескольких операторов по строкам:

```
with (
    open('text_data.txt', 'r+', encoding='utf-8') as f1,
    open('bin_data', 'rb') as f2,
    open('data.txt', 'r', encoding='utf-8',
errors='backslashreplace') as f3
):
    print(list(f1))
    print(list(f2))
    print(list(f3))
```

Чтение файла: целиком, через генератор

Рассмотрим подробнее варианты чтения информации из файла.

- **Чтение в список**

```
with open('text_data.txt', 'r', encoding='utf-8') as f:
    print(list(f))
```

Мы уже использовали приём чтения всего файла построчно в ячейки списка. Медленная по времени и затратная по памяти операция. Но обратите внимание на окончание каждой строки. Символ “\n” означает перенос строки. Разные ОС для обозначения переноса строки используют “\n”, “\r\n” или даже “\r”. Python берёт работу по конвертации символа окончания строки на себя. При чтении любое окончание заменяется на “\n”. А при записи текстового файла окончание “\n” заменяется на окончание для вашей ОС.

- **Чтение методом `read`**

Ещё один вариант чтения файла — метод `read()`.

`read(n=-1)` — читает `n` символов или `n` байт информации из файла. Если `n` отрицательное или не указана, читает весь файл. Попытка чтения будет даже в том случае, когда файл больше оперативной памяти.

```
with open('text_data.txt', 'r', encoding='utf-8') as f:
    res = f.read()
    print(f'Читаем первый раз\n{res}')
    res = f.read()
    print(f'Читаем второй раз\n{res}')
```

Если прочитать файл до конца, повторные попытки чтения не будут вызывать ошибку. Метод будет возвращать пустую строку.

Также при чтении через `read` не добавляются символы переноса строки. Точнее мы не видим “\n”, а видим перенос строки на новую строку.

```
SIZE = 100
with open('text_data.txt', 'r', encoding='utf-8') as f:
    while res := f.read(SIZE):
        print(res)
```

При чтении файла блоками фиксированного размера можно воспользоваться циклом `while`. Дочитав до конца в переменную попадёт пустая строка, которая в цикле будет интерпретирована как ложь и завершит тело цикла.

- **Чтение методом `readline`**

Для чтения текстового файла построчно используют метод `readline`.

```
with open('text_data.txt', 'r', encoding='utf-8') as f:
    while res := f.readline():
        print(res)
```

Обратите внимание на вывод информации. Между строками остаётся по пустой строке. При чтении `readline` возвращает строку с символом переноса на конце. Функция `print` выводит их на печать и автоматически добавляет переход на следующую строку. Получаем пару “\n\n”.

```
SIZE = 100
with open('text_data.txt', 'r', encoding='utf-8') as f:
    while res := f.readline(SIZE):
        print(res)
```

Передача положительного числа в качестве аргумента задаёт границу для длины строки. Если строка короче границы, она возвращается целиком. А если больше, то возвращается часть строки заданного размера. При этом следующий вызов метода вернёт продолжение строки снова фиксированного размера или остаток до конца, если она короче.

- **Чтение циклом for**

Вместо метода `readline` без аргумента можно использовать более короткую запись с циклом `for`

```
with open('text_data.txt', 'r', encoding='utf-8') as f:
    for line in f:
        print(line, end='')
```

Файл построчно попадает в переменную `line`. А для того чтобы избавиться от пустых строк отключили перенос строки в функции `print`.



Важно! Символ переноса строки сохранился в конце каждой строки. Если вам необходимо обработать строку без переносов, можно использовать срезы `line[:-1]` или метод замены `line.replace('\n', '')`

```
SIZE = 100
with open('text_data.txt', 'r', encoding='utf-8') as f:
    for line in f:
        print(line[:-1])
        print(line.replace('\n', ''))
```

Запись и добавление в файл

С режимами записи мы уже познакомились.

- `w` — создаём новый пустой файл для записи. Если файл существует, открываем его для записи и удаляем данные, которые в нём хранились.
- `x` — создаём новый пустой файл для записи. Если файл существует, вызываем ошибку.
- `a` — открываем существующий файл для записи в конец, добавления данных. Если файл не существует, создаём новый файл и записываем в него.

- **Запись методом write**

Метод write принимает на вход строку или набор байт в зависимости от того как вы открыли файл. После записи метод возвращает количество записанной информации.

```
text = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit.'
with open('new_data.txt', 'a', encoding='utf-8') as f:
    res = f.write(text)
    print(f'{res = }\n{len(text) = }')
```

Метод не добавляет символ перехода на новую строку. Если произвести несколько записей, они “склеиваются” в файле.

```
text = ['Lorem ipsum dolor sit amet, consectetur adipiscing elit.',
        'Consequatur debitis explicabo laboriosam sint suscipit temporibus veniam?',
        'Accusantium alias amet fugit iste neque non odit quia saepe totam velit?']
with open('new_data.txt', 'a', encoding='utf-8') as f:
    for line in text:
        res = f.write(line)
        print(f'{res = }\n{len(line) = }')
```

Если каждая строка должна сохраняться в файле с новой строки, необходимо самостоятельно добавить символ переноса - \n

```
text = ['Lorem ipsum dolor sit amet, consectetur adipiscing elit.',
        'Consequatur debitis explicabo laboriosam sint suscipit temporibus veniam?',
        'Accusantium alias amet fugit iste neque non odit quia saepe totam velit?']
with open('new_data.txt', 'a', encoding='utf-8') as f:
    for line in text:
        res = f.write(f'{line}\n')
        print(f'{res = }\n{len(line) = }')
```

- **Запись методом writelines**

Метод writelines принимает в качестве аргумента последовательность и записывает каждый элемент в файл. Элементы последовательности должны быть строками или байтами в зависимости от режима записи.

В отличие от write этот метод ничего не возвращает.

```
text = ['Lorem ipsum dolor sit amet, consectetur adipiscing  
elit.',  
        'Consequatur debitis explicabo laboriosam sint suscipit  
temporibus veniam?',  
        'Accusantium alias amet fugit iste neque non odit quia  
saepe totam velit?', ]  
with open('new_data.txt', 'a', encoding='utf-8') as f:  
    f.writelines('\n'.join(text))
```

Для того чтобы каждый элемент списка text сохранялся в файле с новой строки воспользовались строковым методом join. writelines не добавляет переноса между элементами последовательности.

- **print в файл**

Функция print по умолчанию выводит информацию в поток вывода. Обычно это консоль. Но можно явно передать файловый объект для печати в файл. Для этого надо воспользоваться ключевым параметром file.

```
text = ['Lorem ipsum dolor sit amet, consectetur adipiscing  
elit.',  
        'Consequatur debitis explicabo laboriosam sint suscipit  
temporibus veniam?',  
        'Accusantium alias amet fugit iste neque non odit quia  
saepe totam velit?', ]  
with open('new_data.txt', 'a', encoding='utf-8') as f:  
    for line in text:  
        print(line, file=f)
```

В отличие от методов записи в файл, функция print добавляет перенос строки. Кроме того ничто не мешает явно изменить параметр end функции.

```
text = ['Lorem ipsum dolor sit amet, consectetur adipiscing  
elit.',  
        'Consequatur debitis explicabo laboriosam sint suscipit  
temporibus veniam?',  
        'Accusantium alias amet fugit iste neque non odit quia  
saepe totam velit?', ]  
with open('new_data.txt', 'a', encoding='utf-8') as f:  
    for line in text:  
        print(line, end='***\n##', file=f)
```

Методы перемещения в файле

При работе с файлом можно управлять положением файлового объекта в открытом файле. Действия напоминают движение курсора в строке стрелками влево и вправо.

- **Метод tell**

Метод tell возвращает текущую позицию в файле.

```
text = ['Lorem ipsum dolor sit amet, consectetur adipisicing
elit.',
        'Consequatur debitis explicabo laboriosam sint suscipit
temporibus veniam?',
        'Accusantium alias amet fugit iste neque non odit quia
saepe totam velit?']
with open('new_data.txt', 'w', encoding='utf-8') as f:
    print(f.tell())
    for line in text:
        f.write(f'{line}\n')
        print(f.tell())
    print(f.tell())
print(f.tell()) # ValueError: I/O operation on closed file.
```

Для пустого файла возвращается ноль — начало файла. По мере записи или чтения информации позиция сдвигается к концу файла.

Метод используется для определения в каком месте файла будет произведено чтение или запись.

- **Метод seek**

Метод seek позволяет изменить положение “курсора” в файле.

seek(offset, whence=0), где offset — смещение относительно опорной точки, whence - способ выбора опорной точки.

- whence = 0 - отсчёт от начала файла
- whence = 1 - отсчёт от текущей позиции в файле
- whence = 2 - отсчёт от конца файла



Важно! Значения 1 и 2 допустимы только для работы с бинарными файлами. Исключение seek(0, 2) для перехода в конец текстового файла.

Метод возвращает новую позицию “курсора”.

```
last = before = 0
text = ['Lorem ipsum dolor sit amet, consectetur adipiscing
elit.',
        'Consequatur debitis explicabo laboriosam sint suscipit
temporibus veniam?',
        'Accusantium alias amet fugit iste neque non odit quia
saepe totam velit?', ]
with open('new_data.txt', 'r+', encoding='utf-8') as f:
    while line := f.readline():
        last, before = f.tell(), last
    print(f'{last = }, {before = }')
    print(f'{f.seek(before, 0) = }')
    f.write('\n'.join(text))
```

В примере выше мы открыли текстовый файл для одновременного чтения и записи. Переменные `last` и `before` хранят позиции двух последних прочитанных строк. Дочитав файл в цикле `while` до конца изменяем позицию “курсора” на начало последней строки и начинаем запись. Таким образом мы сохранили все строки файла кроме последней и записали новый текст в конец.

- **Метод `truncate`**

`truncate(size=None)` — метод изменяет размер файла. Если не передать значение в параметр `size` будет удалена часть файла от текущей позиции до конца. Метод возвращает позицию после изменения файла.

```
last = before = 0
with open('new_data.txt', 'r+', encoding='utf-8') as f:
    while line := f.readline():
        last, before = f.tell(), last
    print(f.seek(before, 0))
    print(f.truncate())
```

Если передать параметр `size` метод изменяет размер файла до указанного числа символов или байт от начала файла.

```
size = 64
with open('new_data.txt', 'r+', encoding='utf-8') as f:
    print(f.truncate(size))
```

Если `size` меньше размера файла, происходит усечение файла. Если `size` больше размера файла, он увеличивается до указанного размера. При этом добавленный

размер обычно заполняется нулевыми байтами. Заполнение зависит от конкретной ОС.

Задание

Перед вами несколько строк кода. Что будет храниться в файле после завершения работы программы? И что будет выведено на печать? У вас 3 минуты.

```
start = 10
stop = 100
with open('data.bin', 'bw+') as f:
    for i in range(start, stop + 1):
        f.write(str(i).encode('utf-8'))
        if i % 3 == 0:
            f.seek(-2, 1)
    f.truncate(stop)
    f.seek(0)
    res = f.read(start)
    print(res.decode('utf-8'))
```

2. Файловая система

Далее рассмотрим работу с файловой системой средствами Python. В этой части разберём некоторые модули стандартной библиотеки для решения задач работы с файловой системой.

1. Модуль `os` позволяет использовать функции, зависящие от файловой системы.
2. Для работы с путями используют модуль `os.path`.
3. Модуль `pathlib` позволяет работать с путями файловой системы в ООП стиле.
4. Модуль `shutil` обеспечивает высокоуровневые операции над файлами и группами файлов.

Если задачу можно решить несколькими способами, рассмотрим варианты одновременно.

Работа с каталогами

Операции по работе с файлами и папками должны иметь полный, абсолютный путь к объекту начиная от корневой директории. Или же можно задать относительный путь. Так мы делали в первой части лекции. Файл с кодом и файлы для чтения и записи находились в одном каталоге.

- **Текущий каталог**

Для получения информации о текущем каталоге можно использовать модуль `os` или `pathlib`

```
import os
from pathlib import Path

print(os.getcwd())
print(Path.cwd())
```

Обычно текущим является каталог из которого был запущен `python`.

Для изменения текущего каталога можно воспользоваться функцией `os.chdir`. Она принимает на вход абсолютный или относительный путь до нового текущего каталога.

```
import os
from pathlib import Path

print(os.getcwd())
print(Path.cwd())
os.chdir('../..')
print(os.getcwd())
print(Path.cwd())
```

- **Создание каталогов**

Для создания каталога снова можно воспользоваться двумя модулями

```
import os
from pathlib import Path
```

```
os.mkdir('new_os_dir')
Path('new_path_dir').mkdir()
```

Представленный код создаёт каталог в текущей директории. А если необходимо создать несколько вложенных друг в друга каталогов, код будет следующим:

```
import os
from pathlib import Path

os.makedirs('dir/other_dir/new_os_dir')
Path('some_dir/dir/new_path_dir').mkdir() # FileNotFoundError
Path('some_dir/dir/new_path_dir').mkdir(parents=True)
```

Модуль `os` предоставляет другую функцию — `makedirs`. Модуль `path` работает с тем же методом `mkdir`. Но если в цепочке каталогов будет несуществующий, получим ошибку `FileNotFoundError`. Дополнительный параметр `parents=True` указывает на необходимость создать всех недостающих родительских каталогов.

- **Удаление каталогов**

Для удаления одного каталога подойдут следующие функция и метод

```
import os
from pathlib import Path

# os.rmdir('dir') # OSError
# Path('some_dir').rmdir() # OSError
os.rmdir('dir/other_dir/new_os_dir')
Path('some_dir/dir/new_path_dir').rmdir()
```



Важно! Удалить можно лишь пустой каталог. Если внутри удаляемого каталога есть другие каталоги или файлы, возникнет ошибка `OSError`.

Обратите внимание, что при передаче цепочки каталогов удаляется один, последний из перечисленных. Родительские каталоги остаются без изменений.

Если необходимо удалить каталог со всем его содержимым (вложенные каталоги и файлы), подойдёт функция из модуля `shutil`

```
import shutil

shutil.rmtree('dir/other_dir')
shutil.rmtree('some_dir')
```

В первом случае будет удалена директория `other_dir` со всем содержимым. Директория `dir` останется на месте.

Во втором случае удаляется каталог `some_dir` и его содержимое.

- **Формирование пути**

В операционной системе Windows для указания пути используется обратный слеш `\`. В Unix системах путь разделяется слешем. Чтобы программа работала одинаково на любой ОС рекомендуется использовать специальную функцию `join` из `os.path` для склеивания путей.

Модуль `pathlib` использует более понятный приём с переопределением операции деления.

```
import os
from pathlib import Path

file_1 = os.path.join(os.getcwd(), 'dir', 'new_file.txt')
print(f'{file_1 = }\n{file_1}')

file_2 = Path().cwd() / 'dir' / 'new_file.txt'
print(f'{file_2 = }\n{file_2}')
```

Оба варианта используют разные способы получения результата и хранения информации. Но результат мы получили один и тот же — путь до файла `new_file.txt` в каталоге `dir` текущего каталога.

Чтение данных о каталогах

Для получения информации о том какие директории и файлы находятся в текущем каталоге можно воспользоваться следующими вариантами кода.

```
import os
from pathlib import Path

print(os.listdir())

p = Path(Path().cwd())
for obj in p.iterdir():
    print(obj)
```

Функция `listdir` возвращает список файлов и каталогов. Метод `iterdir` у экземпляра класса `Path` является генератором. В цикле он возвращает объекты из выбранной директории.

- **Проверка на директорию, файл и ссылку**

Получив информацию о содержимом текущего каталога зачастую требуется уточнить что перед нами. В каталогах можно хранить другие каталоги и файлы. В файлах содержатся данные. А символьные ссылки указывают на каталоги и файлы из других мест.

Рассмотрим варианты получения информации об объектах, полученных в примере кода выше.

```
import os
from pathlib import Path

dir_list = os.listdir()
for obj in dir_list:
    print(f'{os.path.isdir(obj) = }', end='\t')
    print(f'{os.path.isfile(obj) = }', end='\t')
    print(f'{os.path.islink(obj) = }', end='\t')
    print(f'{obj = }')

p = Path(Path().cwd())
for obj in p.iterdir():
    print(f'{obj.is_dir() = }', end='\t')
    print(f'{obj.is_file() = }', end='\t')
    print(f'{obj.is_symlink() = }', end='\t')
    print(f'{obj = }')
```

Обратите внимание, что при работе с модулем `os` мы получаем объекты `str` типа. Строки передаются в другие функции для получения результата.

При работе с `pathlib` мы итерируемся по объектам `Path`. А если быть точным по экземплярам класса `Path` той операционной системы, в которой работает код. Методы проверки принадлежности являются частью экземпляра.

- **Обход папок через `os.walk()`**

Функция `os.walk` рекурсивно обходит каталоги от переданного в качестве аргумента до самого нижнего уровня вложенности.

```
import os

for dir_path, dir_name, file_name in os.walk(os.getcwd()):
    print(f'{dir_path = }\n{dir_name = }\n{file_name = }\n')
```

Функция возвращает кортеж из трёх значений:

- `dir_path` — абсолютный путь до каталога
- `dir_names` — список с названиями всех каталогов, находящихся в `dir_path`

➤ `dir_names` — список с названиями всех файлов, находящихся в `dir_path`

Вывод продолжается до тех пор пока не будет возвращена информация обо всех директориях, т.е. каждая директория из `dir_names` передаётся в `os.walk` и оказывается в `dir_path`.

Работа с файлами

Рассмотрим базовые операции по работе с файлами как с отдельными объектами.

- **Переименование файлов**

Переименование файла подразумевает сохранение неизменным файла в файловой таблице и в содержимом файла, но изменение его имени. Для переименования можно воспользоваться следующим кодом.

```
import os
from pathlib import Path

os.rename('old_name.py', 'new_name.py')

p = Path('old_file.py')
p.rename('new_file.py')

Path('new_file.py').rename('newest_file.py')
```

Функция `rename` принимает на вход два обязательных аргумента — исходное имя файла и новое имя. Важно чтобы файл существовал, а новое имя не было занято. Модуль `pathlib` позволяет сделать переименование через создание экземпляра класса `Path` с указанием исходного файла. А затем вызов метода `rename` задаёт новое имя. При этом операции могут быть объединены в одну строку через точечную нотацию и без явного сохранения экземпляра класса в переменной.

- **Перемещение файлов**

Перемещение файла подразумевает изменение его позиции в каталоге файловой системы. В процессе перемещения файл может быть переименован.

```
import os
from pathlib import Path
```

```
os.replace('newest_file.py', os.path.join(os.getcwd(), 'dir',
'new_name.py'))

old_file = Path('new_name.py')
new_file = old_file.replace(Path.cwd() / 'new_os_dir' / old_file)
```

Для исходного файла явно не указывали директорию, где он расположен. При переносе была указана текущая директория как отправная точка. Оба варианта имеют одинаковый эффект.

Также при работе через `os` мы присвоили файлу новое имя. А при использовании модуля `pathlib` указали, что после переноса файл должен сохранить старое имя.

• Копирование файлов

Для копирования файлов лучше всего подходит модуль `shutil`, который предоставляет ряд высокоуровневых операций.

```
import shutil

shutil.copy('one.txt', 'dir')
shutil.copy2('two.txt', 'dir')
```

Функции `copy` и `copy2` работают схожим образом. Они принимают файл для копирования и целевой каталог. Если такого каталога не существует, функция попытается присвоить копируемому файлу имя “цели”.

Отличие состоит в том, что функция `copy2` помимо содержимого файла пытается скопировать и связанные с ним метаданные.

Если стоит задача скопировать каталог со всем его содержимым в новое место, модуль предоставляет функции `copytree`.

```
import shutil

shutil.copytree('dir', 'one_more_dir')
```

Функция `copytree` рекурсивно обходит указанный каталог и копирует его содержимое в новое место.

• Удаление файлов

Вариант удаления всего каталога целиком с содержимым мы уже рассматривали сегодня.

```
import shutil
```



```
shutil.rmtree('dir')
```

Если же необходимо удалить один файл, можно воспользоваться следующими вариантами:

```
import os
from pathlib import Path

os.remove('one_more_dir/one.txt')
Path('one_more_dir/one_more.txt').unlink()
```

И функция `remove` и метод `unlink` пытаются удалить файл по указанному пути.

Задание

Перед вами несколько строчек кода. Какие каталоги и файлы будут созданы после его выполнения? У вас три минуты.

```
import os
import shutil
from pathlib import Path

for i in range(10):
    with open(f'file_{i}.txt', 'w', encoding='utf-8') as f:
        f.write('Hello world!')
os.mkdir('new_dir')
for i in range(2, 10, 2):
    f = Path(f'file_{i}.txt')
    f.replace('new_dir' / f)
shutil.copypath('new_dir', Path.cwd() / 'dir_new')
```

Вывод

На этой лекции мы:

1. Разобрались в особенностях работы с файлами и каталогами в Python

2. Изучили функцию `open` для работы с содержимым файла
3. Узнали о возможностях стандартной библиотеки для работы с файлами и каталогами

Краткий анонс следующей лекции

1. Разберёмся в сериализации и десериализации данных
2. Изучим самый популярный формат сериализации — JSON
3. Узнаем о чтении и записи таблиц в формате CSV
4. Разберёмся с внутренним сериализатором Python — модулем `pickle`

Домашнее задание

1. Загляните в документацию к Python и изучите особенности и нюансы работы с файлами и каталогами в рассматриваемых на уроке модулях.

Оглавление

На этой лекции мы	2
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции Сериализация данных	3
1. JSON	4
Преобразование JSON в Python	6
Преобразование Python в JSON	7
Дополнительные параметры dump и dumps	9
Задание	10
2. CSV	11
Формат CSV	11
Модуль CSV	12
Чтение CSV	12
Запись CSV	13
Чтение в словарь	15
Запись из словаря	15
Задание	17
3. Pickle	17
Допустимые типы данных для преобразования	18
Десериализация	20
Задание	21
Вывод	21

На этой лекции мы

1. Разберёмся в сериализации и десериализации данных
2. Изучим самый популярный формат сериализации - JSON
3. Узнаем о чтении и записи таблиц в формате CSV
4. Разберёмся с внутренним сериализатором Python - модулем pickle

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались в особенностях работы с файлами и каталогами в Python
2. Изучили функцию `open` для работы с содержимым файла
3. Узнали о возможностях стандартной библиотеки для работы с файлами и каталогами

Термины лекции

- **Сериализация** — это процесс преобразования объекта в поток байтов для сохранения или передачи в память, базу данных или файл.
- **Десериализация** — восстановление объектов из байт, сохранение которых было произведено ранее. Процедура выгрузки «зафиксированной» информации пользователем.

Подробный текст лекции

Сериализация данных

Под термином сериализация принято понимать преобразование объектов, а точнее хранящихся в них данных в набор байт для пересылки или хранения. При этом сериализованные данные можно подвергнуть десериализации, т.е. восстановлению данных из байт в исходные объекты. Особенностью сериализации является превращение сложных многоуровневых объектов различной степени вложенности в линейный набор байт. Такой поток легко передавать по каналам связи, хранить в файле и т.п.

Далее на лекции рассмотрим несколько популярных форматов данных, которые используются не только в Python, но и в современном программировании. Отдельно коснёмся сериализации специфичной только для Python.

1. JSON

JSON (JavaScript Object Notation) — это популярный формат текстовых данных, который используется для обмена данными в современных веб- и мобильных приложениях. Кроме того, JSON используется для хранения неструктурированных данных в файлах журналов или базах данных NoSQL.

Не смотря на JavaScript в названии формат JSON не привязывается к конкретному ЯП. Созданный на Python JSON может быть прочитан приложением под Андроид на Java и под iPhone на Swift. Как результат JSON стал одним из самых популярных форматов передачи данных.

Формат JSON

Рассмотрим пример JSON файла

```
{  
  "id": 2,  
  "name": "Ervin Howell",
```

```

"username": "Antonette",
"email": [
    "Shanna@melissa.tv",
    "antonette@howel.com"
],
"address": {
    "street": "Victor Plains",
    "suite": "Suite 879",
    "city": "Wisokyburgh",
    "zipcode": "90566-7771",
    "geo": {
        "lat": "-43.9509",
        "lng": "-34.4618"
    }
},
"phone": "010-692-6593 x09125",
"website": "anastasia.net",
"company": {
    "name": "Deckow-Crist",
    "catchPhrase": "Proactive didactic contingency",
    "bs": "synergize scalable supply-chains"
}
}

```

Как несложно заметить JSON похож на словарь Python. В целом верно, но есть небольшие отличия. Некоторые типы данных Python не совпадают с типами в формате JSON. Поэтому при конвертации словаря в JSON и последующей конвертации в словарь типы данных могут оказаться иными.

Python	JSON	Python
dict	object	dict
list, tuple	array	list
str	string	str
int	number (int)	int
float	number (real)	float
True	true	True
False	false	False
None	null	None

Обратите внимание, что list и tuple конвертируется в массив array, а при обратной конвертации получаем только list.

Если мы храним несколько JSON объектов, в Python обычно используют list. Для JSON в этом случае используется array — в квадратных скобках записываются JSON объекты разделённые запятыми.

Модуль JSON

Для работы с форматом в Python есть встроенный модуль json. Для его использования достаточно импорта в начале файла:

```
import json
```

Рассмотрим четыре основных функции модуля.

- **Преобразование JSON в Python**

Договоримся, что представленный выше объект JSON сохранён в виде текстового файла user.json в кодировке UTF-8 в той же директории, что и исполняемый код.

```
import json

with open('user.json', 'r', encoding='utf-8') as f:
    json_file = json.load(f)

print(f'{type(json_file) = }\n{json_file = }')
print(f'{json_file["name"] = }')
print(f'{json_file["address"]["geo"] = }')
print(f'{json_file["email"] = }')
```

Функция load загрузила объект из файла и произвела его десериализацию — превращение в словарь dict. Дальнейшие манипуляции со словарём не вызовут затруднений у Python разработчиков.

А теперь представим, что мы подготовили информацию в виде многострочного str в python и хотим превратить его из JSON строки в объекты Python.

```
import json

json_text = """
[
    {
        "userId": 1,
```

```

        "id": 9,
        "title": "nesciunt iure omnis dolorem tempora et accusantium",
        "body": "consectetur animi nesciunt iure dolore"
    },
    {
        "userId": 1,
        "id": 10,
        "title": "optio molestias id quia eum",
        "body": "quo et expedita modi cum officia vel magni"
    },
    {
        "userId": 2,
        "id": 11,
        "title": "et ea vero quia laudantium autem",
        "body": "delectus reiciendis molestiae occaecati non minima eveniet qui voluptatibus"
    },
    {
        "userId": 2,
        "id": 12,
        "title": "in quibusdam tempore odit est dolorem",
        "body": "praesentium quia et ea odit et ea voluptas et"
    }
]"""

print(f'{type(json_text) = }\n{json_text = }')
json_list = json.loads(json_text)
print(f'{type(json_list) = }\t{len(json_list) = }\n{json_list = }')

```

Функция `loads` принимает на вход строку хранящуюся как структуру JSON и преобразует её к нужным типам. В нашем примере получили список `list` с четырьмя словарями внутри.

Запомнить различия между функциями просто. Окончание `s` у `loads` намекает на строку. А `load` требует объект с методом `read` для чтения информации. Напомним, что файловый дескриптор имеет метод `read` для чтения информации из файла.



Важно! При открытии файлов важно учитывать их размер. Огромные JSON объекты даёт высокую нагрузку на процессор и оперативную память.

• Преобразование Python в JSON

Что делать, если мы хотим превратить Словарь Python в JSON объект? Для этого используем функции сериализации `dump` и `dumps`. Смысл окончания `s` у `dumps` такой же как и у `loads`.

```
import json

my_dict = {
    "first_name": "Джон",
    "last_name": "Смит",
    "hobbies": ["кузнечное дело", "программирование",
"путешествия"],
    "age": 35,
    "children": [
        {
            "first_name": "Алиса",
            "age": 5
        },
        {
            "first_name": "Маруся",
            "age": 3
        }
    ]
}

print(f'{type(my_dict) = }\n{my_dict = }')
with open('new_user.json', 'w') as f:
    json.dump(my_dict, f)
```

Создаём словарь, открываем файл для записи и сохраняем содержимое функцией `dump`. В качестве первого аргумента функция получает словарь или список, вторым передаем файловый дескриптор либо другой объект, поддерживающий метод `write`. Сама функция `dump` ничего не возвращает.

Если мы откроем файл `new_user.json`, увидим одну длинную строку:

```
{ "first_name":      "\u0414\u0436\u043e\u043d",      "last_name":
"\u0421\u043c\u0438\u0442",      "hobbies":
["\u043a\u0443\u0437\u043d\u0435\u0447\u043d\u043e\u0435 \u0434\u0435\u043b\u043e",
"\u043f\u0440\u043e\u0433\u0440\u0430\u043c\u043c\u0438\u0440\u043e\u0432\u0430\u043d\u0438\u0435",
"\u043f\u0443\u0442\u0435\u0448\u0435\u0441\u0442\u0432\u0438\u044f"],
"age":      35,      "children":      [{"first_name":
"\u0410\u043b\u0438\u0441\u0430",      "age":      5},
{"first_name":
```

```
"\u041c\u0430\u0440\u0443\u0441\u044f", "age": 3}}}
```

Как вы видите символы отличные от ASCII были заменены на специальные коды. Это не означает, что файл повреждён или что-то пошло не так. Проведём десериализацию уже знакомым способом и проверим целостность данных.

```
import json

with open('new_user.json', 'r', encoding='utf-8') as f:
    new_dict = json.load(f)
print(f'{new_dict = }')
```

Если же мы хотим отказаться от символов экранирования в JSON файле, следует установить дополнительный параметр `ensure_ascii` в значение `ложь`.

```
with open('new_user.json', 'w', encoding='utf-8') as f:
    json.dump(my_dict, f, ensure_ascii=False)
```

Воспользуемся словарём `my_dict` ещё раз для проверки функции `dumps`

```
import json

my_dict = {
    "first_name": "Джон",
    "last_name": "Смит",
    "hobbies": ["кузнечное дело", "программирование",
"путешествия"],
    "age": 35,
    "children": [
        {
            "first_name": "Алиса",
            "age": 5
        },
        {
            "first_name": "Маруся",
            "age": 3
        }
    ]
}

dict_to_json_text = json.dumps(my_dict)
print(f'{type(dict_to_json_text) = }\n{dict_to_json_text = }')
```

На выходе получаем объект типа `str` хранящий структуру `json`. Подобные данные мы видели в сохранённом файле.

- **Дополнительные параметры dump и dumps**

Функции для сериализации объектов в JSON поддерживают несколько дополнительных параметров. Они позволяют сделать полученные объекты более удобочитаемыми для пользователя. Разберём на примере функции dumps. Но стоит помнить, что функция dump обладает такими же параметрами с тем же смыслом.

```
import json

my_dict = {
    "id": 123,
    "name": "Clementine Bauche",
    "username": "Cleba",
    "email": "cleba@corp.mail.ru",
    "address": {
        "street": "Central",
        "city": "Metropolis",
        "zipcode": "123456"
    },
    "phone": "+7-999-123-45-67"
}

res = json.dumps(my_dict, indent=2, separators=(',', ':'),
sort_keys=True)
print(res)
```

- Параметр indent отвечает за форматирование с отступами. Теперь JSON выводится не в одну строку, а в несколько. Читать стало удобнее, но размер увеличился.
- Параметр separators принимает на вход кортеж из двух строковых элементов. Первый — символ разделитель элементов. По умолчанию это запятая и пробел. Второй — разделитель ключа и значения. По умолчанию это двоеточие и пробел. Передав запятую и двоеточие без пробела JSON стал компактнее.
- Параметр sort_keys отвечает за сортировку ключей по алфавиту. Нужна сортировка или нет, решать только вам.

Задание

Перед вами несколько строк кода. Какой объект будет получен после его выполнения? У вас три минуты.

```
import json

a = 'Hello world!'
b = {key: value for key, value in enumerate(a)}

c = json.dumps(b, indent=3, separators=('; ', '= '))
print(c)
```

2. CSV

CSV (от англ. Comma-Separated Values — значения, разделённые запятыми) — текстовый формат, предназначенный для представления табличных данных. Строка таблицы соответствует строке текста, которая содержит одно или несколько полей, разделённых запятыми.

CSV часто используют там, где удобно хранить информацию в таблицах. Выгрузки из баз данных, из электронных таблиц для анализа данных.

Формат CSV

При работе с CSV стоит помнить о том, что формат не до конца стандартизирован. Например запятая как символ разделитель может являться частью текста. Чтобы не учитывать такие запятые, можно использовать кавычки. Но тогда кавычки не могут быть частью строки. Кроме того десятичная запятая используется для записи вещественных чисел в некоторых странах. Все эти особенности необходимо учитывать при работе с CSV файлами.

Рассмотрим тестовый CSV файл biostats.csv

```
"Name","Sex","Age","Height (in)","Weight (lbs)"
"Alex","M",41,74,170
"Bert","M",42,68,166
"Carl","M",32,70,155
"Dave","M",39,72,167
"Elly","F",30,66,124
"Fran","F",33,66,115
```

```
"Gwen","F",26,64,121
"Hank","M",30,71,158
"Ivan","M",53,72,175
"Jake","M",32,69,143
"Kate","F",47,69,139
"Luke","M",34,72,163
"Myra","F",23,62,98
"Neil","M",36,75,160
"Omar","M",38,70,145
"Page","F",31,67,135
"Quin","M",29,71,176
"Ruth","F",28,65,131
```

В первой строке могут содержаться заголовки столбцов как в нашем случае. Строки со второй и до конца файла представляют записи. Одна строка — одна запись. Текстовая информация заключена в кавычки, а числа указаны без них.

Подобные текстовые CSV файлы легко получить выгрузив данные из Excel или другой электронной таблицы указав нужный формат. По сути CSV является промежуточным файлом между Excel и Python.

Модуль CSV

Для работы с форматом в Python есть встроенный модуль csv. Для его использования достаточно импорта в начале файла:

```
import csv
```

Рассмотрим основные функции модуля.


- **Чтение CSV**

Функция csv.reader принимает на вход файловый дескриптор и построчно читает информацию.

```
import csv

with open('biostats.csv', 'r', newline='') as f:
    csv_file = csv.reader(f)
    for line in csv_file:
        print(line)
```

```
print(type(line))
```

 **Важно!** При работе с CSV необходимо указывать параметр `newline=""` во время открытия файла.

Кроме файлового дескриптора можно передать любой объект поддерживающий итерацию и возвращающий строки. Также функция `reader` может принимать диалект отличный от заданного по умолчанию — `"excel"`. А при необходимости и дополнительные параметры форматирования, если файл имеет свои особенности. Файл `biostats_tab.csv` хранит те же данные, что и файл выше, но вместо разделителя используется символ табуляции. По сути это разновидность TSV — файл с разделителем табуляцией.

"Name"	"Sex"	"Age"	"Height (in)"	"Weight (lbs)"
"Alex"	"M"	41	74	170
"Bert"	"M"	42	68	166
"Carl"	"M"	32	70	155
"Dave"	"M"	39	72	167
"Elly"	"F"	30	66	124
"Fran"	"F"	33	66	115
"Gwen"	"F"	26	64	121
"Hank"	"M"	30	71	158
"Ivan"	"M"	53	72	175
"Jake"	"M"	32	69	143
"Kate"	"F"	47	69	139
"Luke"	"M"	34	72	163
"Myra"	"F"	23	62	98
"Neil"	"M"	36	75	160
"Omar"	"M"	38	70	145
"Page"	"F"	31	67	135
"Quin"	"M"	29	71	176
"Ruth"	"F"	28	65	131

Добавим несколько параметров для его чтения

```
import csv

with open('biostats_tab.csv', 'r', newline='') as f:
    csv_file = csv.reader(f, dialect='excel-tab',
quoting=csv.QUOTE_NONNUMERIC)
    for line in csv_file:
        print(line)
    print(type(line))
```

- `dialect='excel-tab'` — указали диалект с табуляцией в качестве разделителя
- `quoting=csv.QUOTE_NONNUMERIC` — передали встроенную константу, указывающую функции, что числа без кавычек необходимо преобразовать к типу `float`.

• Запись CSV

Для записи данных в файл используют функцию `writer`, которая возвращает объект конвертирующий данные в формат CSV. Функция `writer` принимает файловый дескриптор и дополнительные параметры записи аналогичные параметрам функции `reader`. При этом данные в файл не записываются пока у возвращённого объекта не будет вызван метод `writerow` для записи одной строки или `writerows` для записи нескольких строк.

Рассмотри на примере.

```
import csv

with (
    open('biostats_tab.csv', 'r', newline='') as f_read,
    open('new_biostats.csv', 'w', newline='', encoding='utf-8')
as f_write
):
    csv_read = csv.reader(f_read, dialect='excel-tab',
quoting=csv.QUOTE_NONNUMERIC)
    csv_write = csv.writer(f_write, dialect='excel', delimiter='
', quotechar='|', quoting=csv.QUOTE_MINIMAL)
    all_data = []
    for i, line in enumerate(csv_read):
        if i == 0:
            csv_write.writerow(line)
        else:
            line[2] += 1
            for j in range(2, 4 + 1):
                line[j] = int(line[j])
            all_data.append(line)
    csv_write.writerows(all_data)
```

1. Используя менеджер контекста `with` открыли два файла. Из первого читаем информацию, а второй создаём для записи.
2. Функция `reader` возвращает объект `csv_read` для чтения как в пример выше.
3. Функция `writer` возвращает объект `csv_write` для записи. Мы указали:
 - a. диалект “excel”
 - b. в качестве разделителя столбцов будем использовать пробел

- c. если символ разделитель (пробел) есть в данных, экранируем их вертикальной чертой
 - d. символ экранирования используем по минимуму, только там где он необходим для разрешения конфликта с разделителем
4. В цикле читаем все строки из исходного файл. При этом строку с заголовком сразу записываем методом `writerow`.
 5. Для остальных строк увеличиваем возраст на единицу, преобразуем вещественные числа в целые и сохраняем список в матрицу `all_data`
 6. Одним запросом `writerows(all_data)` сохраняем матрицу в файл.

• Чтение в словарь

Помимо сохранения таблицы в список можно использовать для хранения словарь. Ключи словаря — названия столбцов, значения — очередная строка файла CSV. Прочитаем файл `biostats_tab.csv` из примера выше, но не в список, а в словарь.

Воспользуемся классом `DictReader`.

```
import csv

with open('biostats_tab.csv', 'r', newline='') as f:
    csv_file = csv.DictReader(f, fieldnames=["name", "sex",
"age", "height", "weight", "office"],
                                restkey="new", restval="Main
Office", dialect='excel-tab', quoting=csv.QUOTE_NONNUMERIC)
    for line in csv_file:
        print(f'{line = }')
        print(f'{line["name"] = }\t{line["age"] = }')
```

Если передать список строк в параметр `fieldnames`, они будут использоваться для ключей словаря, а не первая строка файла. В нашем примере передан “лишний” ключ `count`. Т.к. в таблице нет шестого столбца, ему было присвоено значение из параметра `restval`.

```
import csv

with open('biostats_tab.csv', 'r', newline='') as f:
    csv_file = csv.DictReader(f, fieldnames=["name", "sex",
"age", ], restkey="new", restval="Main Office",
                                dialect='excel-tab',
quoting=csv.QUOTE_NONNUMERIC)
    for line in csv_file:
        print(f'{line = }')
        print(f'{line["name"] = }\t{line["age"] = }')
```


Если количество ключей оказывается меньше, чем столбцов, недостающий ключ берётся из параметра `restkey`. При этом все столбцы без ключа сохраняются как элементы списка в `restkey` ключ.

- **Запись из словаря**

Для записи содержимого словаря в CSV используют класс `DictWriter`. Его параметры схожи с рассмотренными выше параметрами `DictReader`.

```
import csv
from typing import Iterator

with (
    open('biostats_tab.csv', 'r', newline='') as f_read,
    open('biostats_new.csv', 'w', newline='', encoding='utf-8')
as f_write
):
    csv_read: Iterator[dict] = csv.DictReader(f_read,
fieldnames=["name", "sex", "age", "height", "weight", "office"],
                                                    restval="Main
Office", dialect='excel-tab', quoting=csv.QUOTE_NONNUMERIC)
    csv_write = csv.DictWriter(f_write, fieldnames=["id", "name",
"office", "sex", "age", "height", "weight"],
                                                    dialect='excel-tab',
quoting=csv.QUOTE_ALL)
    csv_write.writeheader()
    all_data = []
    for i, dict_row in enumerate(csv_read):
        if i != 0:
            dict_row['id'] = i
            dict_row['age'] += 1
            all_data.append(dict_row)
    csv_write.writerows(all_data)
```

Класс `DictWriter` получил список полей для записи, где добавлено новое поле `id`. В качестве диалекта выбран `excel` с табуляцией. В параметре `quoting` указали, что все значения стоит заключать в кавычки.

Новый для нас метод `writeheader` сохранил первую строку с заголовками в том порядке, в котором мы их перечислили в параметре `fieldnames`. Далее мы работаем с элементами словаря и формируем список словарей для одноразовой записи в файл.



Важно! Обратите внимание на импорт объекта `Iterator` из модуля `typing`. При написании кода IDE подсвечивала возможные ошибки, так как не понимала что за объект `csv_read`. Запись `csv_read: Iterator[dict] = ...` сообщает, что мы используем объект итератор, который возвращает словари. После уточнения типа IDE исключила подсветку “ошибок”.

Задание

Перед вами несколько строк кода. Что будет записано в файл после его выполнения? У вас три минуты.

```
import csv

with open('quest.csv', 'w', newline='', encoding='utf-8') as f_write:
    csv_write = csv.DictWriter(f_write, fieldnames=["number",
"number", "data"], restval='Hello world!', dialect='excel',
delimiter='#', quotechar='=', quoting=csv.QUOTE_NONNUMERIC)
    csv_write.writeheader()
    dict_row = {}
    for i in range(10):
        dict_row['number'] = i
        dict_row['name'] = str(i)
        csv_write.writerow(dict_row)
```

3. Pickle

Рассмотренные выше сериализаторы универсальны. Они не привязаны к конкретному языку программирования. А следовательно прочитать и понять файл сможет любой разработчик. Однако сложные структуры данных Python не всегда возможно сохранить в таблице CSV или JSON объекте.

Python предлагает модуль `pickle` для сериализации и десериализации своих структур в поток байт. Преобразования возможны в любом месте и в любое время, если вы используете Python. Но данные окажутся бесполезными, если вы передаёте их для обработки другим ЯП.



Крайне важно! Модуль `pickle` не занимается проверкой потока байт на безопасность перед распаковкой. Не используйте его с тем набором байт, безопасность которого не можете гарантировать.

```
import pickle

res = pickle.loads(b"cos\nsystem\n(S'echo Hello world!'\ntr.")
print(res)
```

В этом безобидном примере модуль получил доступ к консоли и вывел сообщение “Привет, мир!” прежде чем десериализовать поток байт в число ноль.

Допустимые типы данных для преобразования

Модуль `pickle` может обработать следующие структуры Python:

- `None`, `True` и `False`;
- `int`, `float`, `complex`;
- `str`, `bytes`, `bytearrays`;
- `tuple`, `list`, `set`, `dict` если они содержат объекты, обрабатываемые `pickle`;
- встроенные функции и функции созданные разработчиком и доступные из верхнего уровня модуля, кроме `lambda` функций;
- классы доступные из верхнего уровня модуля;
- экземпляры классов, если `pickle` смог обработать их дандер `__dict__` или результат вызова метода `__getstate__()`.

Список достаточно большой, чтобы позволить сериализовывать большую часть Python структур.

Сериализация

Преобразуем словарь из главы про JSON в набор байт средствами модуля `pickle`.

```
import pickle
```

```

my_dict = {
    "first_name": "Джон",
    "last_name": "Смит",
    "hobbies": ["кузнечное дело", "программирование",
"путешествия"],
    "age": 35,
    "children": [
        {
            "first_name": "Алиса",
            "age": 5
        },
        {
            "first_name": "Маруся",
            "age": 3
        }
    ]
}
print(my_dict)
res = pickle.dumps(my_dict, protocol=pickle.DEFAULT_PROTOCOL)
print(f'{res = }')

```

Вызываем функцию `dumps` для преобразования всей структуры в строку байт. Отдельно указали протокол по умолчанию. Модуль `pickle` имеет несколько протоколов, который не гарантируют совместимость с более старыми версиями. Сейчас протоколом по умолчанию является версия 4. Она появилась в Python 3.4, а стала дефолтным протоколом с Python 3.8. Если вы не занимаетесь поддержкой старого кода, можно смело использовать четвёртую версию протокола. Попробуем сохранить объекты, неподдерживаемые JSON в бинарный файл.

```

import pickle

def func(a, b, c):
    return a + b + c

my_dict = {
    "numbers": [42, 4.1415, 7+3j],
    "functions": (func, sum, max),
    "others": {True, False, 'Hello world!'},
}

with open('my_dict.pickle', 'wb') as f:
    pickle.dump(my_dict, f)

```

Функция `dump` принимает на вход объект для сериализации и файловый дескриптор. При этом файл должен быть открыт для записи как бинарный. В целом мы можем заменить файл на любой объект потоковой записи, который реализует метод `write`.

Обратите внимание на то, что мы добавили в словарь функцию `func`, которая принимает на вход три значения и возвращает их сумму. К ней мы вернемся ниже.

Десериализация

Проведём обратные операции. Набор байт мы уже восстанавливали, когда разбирались с безопасностью данных. Уточним детали.

```
import pickle

data = b'\x80\x04\x95\xe3\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\nfirst_name\x94\x8c\x08\xd0\x94\xd0\xb6\xd0\xbe\xd0xbd\x94\x8c\tlast_name\x94\x8c\x08\xd0\xa1\xd0xbc\xd0\xb8\xd1\x82\x94\x8c\x07hobbies\x94]\x94(\x8c\x1b\xd0\xba\xd1\x83\xd0\xb7\xd0\xbd\xd0\xb5\xd1\x87\xd0\xbd\xd0\xbe\xd0\xb5 \xd0\xb4\xd0\xb5\xd0\xbb\xd0\xbe\x94\x8c\xd0\xbf\xd1\x80\xd0\xbe\xd0\xb3\xd1\x80\xd0\xb0\xd0xbc\xd0\xbc\xd0\xb8\xd1\x80\xd0\xbe\xd0\xb2\xd0\xb0\xd0\xbd\xd0\xb8\xd0\xb5\x94\x8c\x16\xd0\xbf\xd1\x83\xd1\x82\xd0\xb5\xd1\x88\xd0\xb5\xd1\x81\xd1\x82\xd0\xb2\xd0\xb8\xd1\x8f\x94e\x8c\x03age\x94K#\x8c\x08children\x94]\x94(\x94(h\x01\x8c\n\xd0\x90\xd0\xbb\xd0\xb8\xd1\x81\xd0\xb0\x94h\nK\x05u}\x94(h\x01\x8c\x0c\xd0\x9c\xd0\xb0\xd1\x80\xd1\x83\xd1\x81\xd1\x8f\x94h\nK\x03ueu.'
```

```
new_dict = pickle.loads(data)
print(f'{new_dict = }')
```

Функция получила на вход набор байт и восстановила из них исходный словарь. Уточним, что `loads` имеет ряд дополнительных параметров: `fix_imports=True`, `encoding='ASCII'`, `errors='strict'`. Они нужны для десериализации объектов созданных в Python 2. А так как поддержка второй версии Python завершена в 2020 году, нет смысла разбирать назначение параметров.

В финале загрузим данные из файла `my_dict.pickle`, который создали ранее.

```
import pickle
```

```
def func(a, b, c):
    return a * b * c

with open('my_dict.pickle', 'rb') as f:
    new_dict = pickle.load(f)
print(f'{new_dict = }')
print(f'{new_dict["functions"][0](2, 3, 4) = }')
```

Содержимое словаря в точности соответствует исходному. Но есть одно но. При вызове функции func, которая лежит в нулевой ячейке кортежа по ключу functions мы получили не сумму трёх чисел, а произведение. В файле, где произведена десериализация есть функция func, которая умножает числа. Модуль pickle указал в словаре её, а не исходную. Более того, если бы функции с нужным именем не было, десериализация завершилась бы ошибкой.

Задание

Перед вами несколько строк кода. Что будет записано в файл после его выполнения? У вас три минуты.

```
import pickle

my_dict = {'numbers': [42, 4.1415, (7 + 3j)],
           'functions': (sum, max),
           'others': {False, True, 'Hello world!'}}

res = pickle.dumps(my_dict)
with open('quest.pickle', 'wb') as f:
    pickle.dump(f'{res = }', f)
```

Вывод

На этой лекции мы:

1. Разобрались в сериализации и десериализации данных
2. Изучили самый популярный формат сериализации — JSON
3. Узнали о чтении и записи таблиц в формате CSV
4. Разобрались с внутренним сериализатором Python — модулем pickle

Краткий анонс следующей лекции

1. Разберём замыкания в программировании
2. Изучим возможности Python по созданию декораторов
3. Узнаем как создавать декораторы с параметрами
4. Разберём работу некоторых декораторов из модуля functools

Домашнее задание

Возьмите код прошлых уроков и попытайтесь сериализовать используемые в нём объекты на основе знаний с сегодняшнего занятия. Попробуйте все варианты сериализации и десериализации в разных форматах.

Оглавление

На этой лекции мы	2
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции	
1. Что такое замыкания	3
Функция как объект высшего порядка	4
Замыкаем изменяемые и неизменяемые объекты	6
Задание	8
2. Простой декоратор без параметров	
Передача функции в качестве аргумента	8
Множественное декорирование	11
Дополнительные переменные в декораторе	13
Задание	14
3. Декоратор с параметрами	14
4. Декораторы functools	17
Декоратор cache	18
Вывод	19

На этой лекции мы

1. Разберём замыкания в программировании
2. Изучим возможности Python по созданию декораторов
3. Узнаем как создавать декораторы с параметрами
4. Разберём работу некоторых декораторов из модуля functools

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались в сериализации и десериализации данных
2. Изучили самый популярный формат сериализации — JSON
3. Узнали о чтении и записи таблиц в формате CSV
4. Разобрались с внутренним сериализатором Python — модулем pickle

Термины лекции

- **Замыкание (англ. closure) в программировании** — функция первого класса, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами.
- **Декоратор** — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту.

Подробный текст лекции

1. Что такое замыкания

Прежде чем погрузиться в декораторы поговорим о замыканиях в программировании вообще и в Python в частности. Плюс стоит вспомнить об областях видимости в Python.

Области видимости

```
def func(a):  
    x = 42  
    res = x + a  
    return res  
  
x = 73  
print(func(64))
```

В этом примере глобальная переменная `x` равна 73, но при сложении внутри функции к значению `a` прибавляется 42 — значение локальной переменной `x`.

Функция как объект высшего порядка

Рассмотрим простой пример функции:

```
def add_str(a: str, b: str) -> str:  
    return a + ' ' + b  
  
print(add_str('Hello', 'world!'))
```

На вход передаём две строки и возвращаем новую из двух старых и пробела посередине. Но функцию можно переписать иначе. Вспомним, что в Python все функции высшего порядка. А это значит, что их можно передавать как объекты в другие функции:

```
from typing import Callable  
  
def add_one_str(a: str) -> Callable[[str], str]:  
    def add_two_str(b: str) -> str:  
        return a + ' ' + b  
  
    return add_two_str  
  
print(add_one_str('Hello')('world!'))
```

Результат получили такой же, но код работает иначе.

- Функция `add_one_str` принимает на вход один параметр в качестве начала строки и возвращает локальную функцию `add_two_str`. Обратите внимание на отсутствие круглых скобок. Функцию передаём, а не вызываем.
- Функция `add_two_str` принимает вторую часть строки, соединяет её с первой и возвращает ответ.
- При вызове функций значения указывается в отдельных круглых скобках. Первое попадает в параметр `a`. Далее часть строки: `add_one_str('Hello')` возвращает функцию `add_two_str` и уже она вызывается и принимает аргумент во вторых скобках.

Благодаря передаче одной функции другой мы можем создавать замыкания.

Замыкаем функцию с параметрами

Внесём небольшие правки в пример кода:

```
from typing import Callable

def add_one_str(a: str) -> Callable[[str], str]:
    def add_two_str(b: str) -> str:
        return a + ' ' + b

    return add_two_str

hello = add_one_str('Hello')
bye = add_one_str('Good bye')

print(hello('world!'))
print(hello('friend!'))
print(bye('wonderful world!'))

print(f'{type(add_one_str)} = {type(add_one_str)}, {add_one_str.__name__} = {add_one_str.__name__}, {id(add_one_str)} = {id(add_one_str)}')
print(f'{type(hello)} = {type(hello)}, {hello.__name__} = {hello.__name__}, {id(hello)} = {id(hello)}')
print(f'{type(bye)} = {type(bye)}, {bye.__name__} = {bye.__name__}, {id(bye)} = {id(bye)}')
```

Во-первых мы не изменяли исходную функцию. Но мы создали две переменные `hello` и `bye` и поместили в них результат работы функции `add_one_str` с разными аргументами. Теперь мы можем вызывать новые функции и получать объединённые

строки передавая только окончание. Первая часть строки оказалась замкнута в локальной области видимости. И у каждой из двух новых функций область своя и начало строки своё.

А теперь посмотрите на результат работы трёх нижних строк кода. Все три переменные являются функциями, что очевидно. Но если функция `add_one_str` является самой собой, то функции `hello` и `bye` на самом деле являются двумя разными экземплярами функции `add_two_str`. `id`, т.е. адреса в оперативной памяти разные, а названия указывают на оригинал.

Замыкаем изменяемые и неизменяемые объекты

В очередной раз внесём правки в пример кода:

```
from typing import Callable

def add_one_str(a: str) -> Callable[[str], str]:
    names = []

    def add_two_str(b: str) -> str:
        names.append(b)
        return a + ', ' + ', '.join(names)

    return add_two_str

hello = add_one_str('Hello')
bye = add_one_str('Good bye')

print(hello('Alex'))
print(hello('Karina'))
print(bye('Alina'))
print(hello('John'))
print(bye('Neo'))
```

Во внешнюю функцию добавлен список `names`. При каждом вызове внутренней функции в список добавляется новое значение из параметра `b` и возвращается полное содержимое списка в виде строки. У каждой из двух функций `hello` и `bye` оказывается свой список `names`. Они не связаны между собой, но каждый хранит

список имён до конца работы программы. Обратите внимание, что `list` является изменяемым типом данных. Что будет, если мы перепишем код и заменим `list` на неизменяемый `str`?

```
from typing import Callable

def add_one_str(a: str) -> Callable[[str], str]:
    text = ''

    def add_two_str(b: str) -> str:
        nonlocal text
        text += ', ' + b
        return a + text

    return add_two_str

hello = add_one_str('Hello')
bye = add_one_str('Good bye')

print(hello('Alex'))
print(hello('Karina'))
print(bye('Alina'))
print(hello('John'))
print(bye('Neo'))
```

Изменения способа получения строки с `join` для списка на конкатенацию для строки не принципиально. Но стоит помнить, что сложение строк более дорогая операция по времени и по памяти, особенно если она находится внутри цикла.

Что более важно - неизменяемый тип данных у строки `text`. Без добавления строчки кода `nonlocal text` была бы получена ошибка `UnboundLocalError: local variable 'text' referenced before assignment`. Мы явно указали, что хотим обращаться к неизменяемому объекту для изменения его значения.

Как можно изменить неизменяемое? Мы создаём новый объект и присваиваем ему старое имя. Старый объект будет удалён сборщиком мусора. А команда `nonlocal` сообщает Python, что изменения ссылки на объект должны затронуть область видимости за пределами функции `add_two_str`.

Подведём промежуточный итог. Благодаря тому что в Python всё объект, а функции являются функциями высшего порядка, мы можем вкладывать во внешнюю функцию различные переменные и внутренние функции. Далее возвращая из внешней функции внутреннюю создаём замыкания.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
from typing import Callable

def main(x: int) -> Callable[[int], dict[int, int]]:
    d = {}

    def loc(y: int) -> dict[int, int]:
        for i in range(y):
            d[i] = x ** i
        return d

    return loc

small = main(42)
big = main(73)
print(small(7))
print(big(7))
print(small(3))
```

2. Простой декоратор без параметров

Передача функции в качестве аргумента

До этого момента наш код возвращал функции, но не принимал их. Исправим ситуацию на примере самописной функции нахождения факториала. Напомним, что факториал числа - произведение чисел от единицы до заданного числа.

```
import time
```

```

from typing import Callable

def main(func: Callable):
    def wrapper(*args, **kwargs):
        print(f'Запуск функции {func.__name__} в {time.time()}')
        result = func(*args, **kwargs)
        print(f'Результат функции {func.__name__}: {result}')
        print(f'Завершение функции {func.__name__} в {time.time()}')
        return result

    return wrapper

def factorial(n: int) -> int:
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f

print(f'{factorial(1000) = }')
control = main(factorial)
print(f'{control.__name__ = }')
print(f'{control(1000) = }')

```

- Функция `main` принимает на вход другую функцию. Внутри функции определена функция `wrapper`, которая возвращается функцией `main`.
- Функция `wrapper` принимает пару параметров `*args` и `**kwargs`. С ними вы уже знакомы. Подобная запись позволяет принять любое число позиционных аргументов и сохранить их в кортеже `args`, а также любое число ключевых аргументов с сохранением в словаре `kwargs`.
Обязательной строкой внутри `wrapper` является `result = func(*args, **kwargs)`. Переданная в качестве аргумента функция `func` вызывается со всеми аргументами, которые были переданы. Дополнительно выводим информацию о времени запуска, результатах и времени завершения работы функции. Не забываем вернуть результат работы `func` из `wrapper`.
- Функция `factorial` вычисляет факториал для заданного числа.
- В нижней части кода запускаем поиск факториала, проверяем работоспособность. Далее мы создаём функцию `control` в которую помещается `wrapper` с замкнутой внутри функций `func` — нашей функцией `factorial`. При вызове контрольной функции помимо результата поиска факториала получаем вывод прописанный внутри `wrapper`.

Замыкание переданной в качестве аргумента функции внутри другой функции называется декорированием функции. В нашем примере `main` — декоратор, которым мы декорировали функцию `factorial`.

Синтаксический сахар Python, @

В языке Python есть более элегантная возможность создания декораторов — синтаксический сахар. Для этого используется символ “@” слитно с именем декоратора. Строка кода пишется непосредственно над определением функции или метода.

```
import time
from typing import Callable

def main(func: Callable):
    def wrapper(*args, **kwargs):
        print(f'Запуск функции {func.__name__} в {time.time()}')
        result = func(*args, **kwargs)
        print(f'Результат функции {func.__name__}: {result}')
        print(f'Завершение функции {func.__name__} в {time.time()}')
        return result

    return wrapper

@main
def factorial(n: int) -> int:
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f

print(f'{factorial(1000) = }')
```

Добавили декоратор `@main` к функции `factorial`. Необходимость в присваивании значения новой переменной отпала. Несколько нижних строк кода из старого примера удалили за ненадобностью. Кроме того мы сохранили старое имя функции.



Важно! Функция декоратор должна быть определена в коде раньше, чем использована. В противном случае получим ошибку `NameError`

Синтаксический сахар упрощает написание кода, но не является обязательным к применению. Однако в случае с передачей функции в замыкание использование символа `@` считается нормой. Связано это с тем, что присваивание переменной нового значения происходит очень часто в коде. И понять создаём мы замыкание функции или присваиваем что-то другое сложно. Когда же речь идёт о присваивании через `@`, сразу ясно что используется декоратор

Множественное декорирование

Python позволяет использовать несколько декораторов на одной функции. Рассмотрим на простом примере.

```
from typing import Callable

def deco_a(func: Callable):
    def wrapper_a(*args, **kwargs):
        print('Старт декоратора A')
        print(f'Запускаю {func.__name__}')
        res = func(*args, **kwargs)
        print(f'Завершение декоратора A')
        return res

    print('Возвращаем декоратор A')
    return wrapper_a

def deco_b(func: Callable):
    def wrapper_b(*args, **kwargs):
        print('Старт декоратора B')
        print(f'Запускаю {func.__name__}')
        res = func(*args, **kwargs)
        print(f'Завершение декоратора B')
        return res

    print('Возвращаем декоратор B')
    return wrapper_b
```

```

def deco_c(func: Callable):
    def wrapper_c(*args, **kwargs):
        print('Старт декоратора C')
        print(f'Запускаю {func.__name__}')
        res = func(*args, **kwargs)
        print(f'Завершение декоратора C')
        return res

    print('Возвращаем декоратор C')
    return wrapper_c

@deco_c
@deco_b
@deco_a
def main():
    print('Старт основной функции')

main()

```

Мы создали три одинаковых декоратора, которые сообщают о начале и завершении работы и о моменте декорирования: A, B, C.

Обратите внимание на порядок декораторов у функции main. Ближайший к функции декоратор A. Декоратор C находится первым в списке, т.е. он максимально удалён от основной функции.

При запуске кода процесс декорирования начинается снизу вверх, с A, далее B и лишь потом C.

Прежде чем выполнить код основной функции запускается код верхнего декоратора C, далее B, в конце нижний A и только потом код функции main. После того как декорированная функция завершила работу и вернула результат декораторы завершают работу в обратном старту порядке, снизу вверх. В зависимости от решаемых задач порядок декорирования может привести к разным результатам.

Дополнительные переменные в декораторе

Мы уже замыкали внутри функции список для хранения переданных имён. Декораторы открывают большие возможности по модификации основной функции. Рассмотрим пример простого кэширующего декоратора.

```
from typing import Callable

def cache(func: Callable):
    _cache_dict = {}

    def wrapper(*args):
        if args not in _cache_dict:
            _cache_dict[args] = func(*args)
        return _cache_dict[args]

    return wrapper

@cache
def factorial(n: int) -> int:
    print(f'Вычисляю факториал для числа {n}')
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f

print(f'{factorial(10) = }')
print(f'{factorial(15) = }')
print(f'{factorial(10) = }')
print(f'{factorial(20) = }')
print(f'{factorial(10) = }')
print(f'{factorial(20) = }')
```

Внутри декоратора `cache` создали пустой словарь `_cache_dict`. При каждом вызове функции `factorial` внутри обёртки `wrapper` происходит проверка. Если переданное для нахождения факториала число не является ключём словаря, создаём соответствующий ключ и в качестве значения присваиваем ему результат вычисления факториала. Когда в словаре есть ключ, декорируемая функция не вызывается, а ответ сразу возвращается из словаря.



Важно! Мы специально исключили параметр `**kwargs` из функции `wrapper`, т.к. это словарь ключевых аргументов. Попытка использования в

качестве ключа словаря `_cache_dict` другого словаря `kwargs` приведёт к ошибке. Ключом может выступать только неизменяемые объекты.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
import random
from typing import Callable

def cache(func: Callable):
    _cache_dict = {}

    def wrapper(*args):
        if args not in _cache_dict:
            _cache_dict[args] = func(*args)
        return _cache_dict[args]

    return wrapper

@cache
def rnd(a: int, b: int) -> int:
    return random.randint(a, b)

print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 10) = }')
```

3. Декоратор с параметрами

До этого мы вкладывали одну функцию в другую для создания замыкания. Если мы хотим передавать в декоратор дополнительные параметры, понадобится третий

уровень вложенности. Рассмотрим пример кода.

```
import time
from typing import Callable

def count(num: int = 1):
    def deco(func: Callable):
        def wrapper(*args, **kwargs):
            time_for_count = []
            result = None
            for _ in range(num):
                start = time.perf_counter()
                result = func(*args, **kwargs)
                stop = time.perf_counter()
                time_for_count.append(stop - start)
            print(f'Результаты замеров {time_for_count}')
            return result

        return wrapper


    return deco

@count(10)
def factorial(n: int) -> int:
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f


print(f'{factorial(1000) = }')
print(f'{factorial(1000) = }')
```

- Внешняя функция count принимает на вход целое число num. Данный параметр будет использован для цикла for.
- Функция deco как и в прошлых примерах принимает декларируемую функцию.
- Внутренняя функция wrapper создаёт список time_for_count для хранения результатов замеров быстродействия.
 - Запускаем цикл for столько раз, сколько мы передали в декоратор: `@count(10)`

- Внутри цикла for замеряем текущее время. Далее выполняем функцию и сохраняем результат в переменную. Замеряем время после окончания работы функции и сохраняем разницу в список.
- После завершения цикла сообщаем результаты из списка time_for_count и возвращаем результат работы декларируемой функции.
- Используя обёртку для факториала делаем 10 замеров и смотрим время на вычисления.

 **Важно!** Последняя строка дублируется не случайно. Каждый из двух запусков делает по 10 замеров. Если бы список time_for_count был создан на уровень выше, в функции deco, произошло бы его замыкание. В результате каждый новый вызов функции factorial дополнял бы уже существующий список, а не создавал бы новые 10 значений.

Декоратор с параметром может принимать любые значения в зависимости от предназначения.

 **Важно!** Для оценки быстродействия кода рекомендуется использовать модуль timeit из “батареек Python”, а не созданный выше декоратор.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
import random
from typing import Callable

def count(num: int = 1):
    def deco(func: Callable):
        counter = []

        def wrapper(*args, **kwargs):
            for _ in range(num):
                result = func(*args, **kwargs)
                counter.append(result)
            return counter
```

```

        return wrapper

    return deco

@count(10)
def rnd(a: int, b: int) -> int:
    return random.randint(a, b)

print(f'{rnd(1, 10) = }')
print(f'{rnd(1, 100) = }')
print(f'{rnd(1, 1000) = }')
```

4. Декораторы functools

Дополнительные возможности декорирования предоставляет модуль `functools` декоратор.

Декоратор wraps

Рассмотрим [код из прошлой главы](#), но добавим строку документации в функцию `factorial`.

```

...
@count(10)
def factorial(n: int) -> int:
    """Returns the factorial of the number n."""
    f = 1
    for i in range(2, n + 1):
        f *= i
    return f

print(f'{factorial(1000) = }')
print(f'{factorial.__name__ = }')
```

```
help(factorial)
```

Вместо ожидаемого вывода документации о функции и её названия получаем информацию об обёртке wrapper:

```
factorial.__name__ = 'wrapper'
Help on function wrapper in module __main__:

wrapper(*args, **kwargs)
```

Чтобы исправить ситуацию, воспользуемся декоратором wraps из functools.

```
import time
from typing import Callable
from functools import wraps

def count(num: int = 1):
    def deco(func: Callable):
        @wraps(func)
        def wrapper(*args, **kwargs):
            time_for_count = []
            ...
```

Декоратор wraps добавляется к функции wrapper, т.е. к самой глубоко вложенной функции. В качестве аргумента wraps должен получить параметр декларируемой функции. Теперь factorial возвращает свои название и строку документации.

Декоратор cache

Рассматривая возможности по замыканию переменных мы создали кэширующий декоратор. В модуле functools есть декоратор cache с подобным функционалом. При необходимости кэширования данных рекомендуется использовать именно его, как более оптимальное решение из коробки.

```
from functools import cache

@cache
def factorial(n: int) -> int:
```



```
print(f'Вычисляю факториал для числа {n}')
```

```
f = 1
for i in range(2, n + 1):
    f *= i
return f
```



```
print(f'{factorial(10) = }')
```

```
print(f'{factorial(15) = }')
```

```
print(f'{factorial(10) = }')
```

```
print(f'{factorial(20) = }')
```

```
print(f'{factorial(10) = }')
```

```
print(f'{factorial(20) = }')
```

Как вы видите только первые вызовы запускают функцию. Повторный вызов с уже передававшимся аргументом обрабатывается декоратором `cache`.

Вывод

На этой лекции мы:

1. Разобрали замыкания в программировании
2. Изучили возможности Python по созданию декораторов
3. Узнали как создавать декораторы с параметрами
4. Разобрали работу некоторых декораторов из модуля `functools`

Краткий анонс следующей лекции

1. Разберёмся с объектно-ориентированным программированием в Python.
2. Изучим особенности инкапсуляции в языке
3. Узнаем о наследовании и механизме разрешения множественного наследования.
4. Разберёмся с полиморфизмом объектов.

Домашнее задание

1. Примените рассматриваемые на лекции декораторы к функциям, созданным на прошлых уроках.
2. Попробуйте создать свои декораторы. Например вы можете написать декоратор, который считает количество вызовов функции.

Оглавление

На этой лекции мы	2
Дополнительные материалы к лекции	2
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции	
1. Основы ООП в Python	4
Атрибуты класса и экземпляров	6
Параметр self	9
Передача аргументов в экземпляр	9
Методы класса	10
2. Инкапсуляция	12
Одно подчёркивание в начале	13
3. Наследование	17
4. Полиморфизм	25

На этой лекции мы

1. Разберёмся с объектно-ориентированным программированием в Python.
2. Изучим особенности инкапсуляции в языке
3. Узнаем о наследовании и механизме разрешения множественного наследования.
4. Разберёмся с полиморфизмом объектов.

Дополнительные материалы к лекции

1. C3 линейаризация

<https://ru.wikipedia.org/wiki/C3-%D0%BB%D0%B8%D0%BD%D0%B5%D0%B0%D1%80%D0%B8%D0%B7%D0%B0%D1%86%D0%B8%D1%8F>

2. MRO Python <https://www.python.org/download/releases/2.3/mro/>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрали замыкания в программировании
2. Изучили возможности Python по созданию декораторов
3. Узнали как создавать декораторы с параметрами
4. Разобрали работу некоторых декораторов из модуля `functools`

Термины лекции

- **Объектно-ориентированное программирование (сокр. ООП)** — методология программирования, основанная на представлении программы в виде совокупности взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.
- **Инкапсуляция** — свойство системы, позволяющее объединить данные и методы, работающие с ними, в классе.
- **Наследование** — свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствованной функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.
- **Полиморфизм** — свойство системы, позволяющее использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.
- **Класс** — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является

моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).

Подробный текст лекции

1. Основы ООП в Python

Как вы помните, что в Python всё объект. Объекты строятся на основе классов. Объекты также называют экземплярами класса. Например выполняя следующий код вы создаёте экземпляр класса list обратившись к классу list

```
data = list((1, 2, 3))
print(f'{data = }, {type(data) = }, {type(list) = }')
```

Тип функции list — type. Перед нами класс языка Python. Привычные функции для создания базовых объектов Python на самом деле являются классами: list, tuple, int, str и т.д.

При разработке языка было введено минимум новых команд и действия для написания кода в ООП стиле похожи на написание кода в функциональном стиле. Отличия и особенности разберём далее.

Классы

Вы уже знаете, что в Python есть несколько областей видимости. Например переменные объявленные внутри функции, являются локальными переменными функции. Так же вы помните о модулях, которые могут хранить в себе переменные и функции. Для обращение к ним используется импорт и точечная нотация:

```
import random
import supermodule

result1 = random.randint(1, 10)
result2 = supermodule.randint(42)
```

При этом разные модули могут содержать одноимённые функции и переменные. Это не будет вызывать ошибок, т.к. пространство имён разграничивает их по областям видимости.

Класс Python это ещё один способ создать локальную область видимости, поместив в неё переменные и функции класса. При этом для обращения к ним так же используется точечная нотация. Зачем же вводить классы, если они ведут себя как модули? Классы позволяют создавать свои экземпляры. Количество экземпляров зависит от решаемой задачи. И у каждого экземпляра будут свои переменные и функции со своими значениями. Каждый экземпляр класса создаёт свою локальную область видимости.

Представим, что мы пишем компьютерную игру. Создадим класс `Person`. Будем рассматривать особенности ООП на его примере в рамках занятия.

```
class Person:
    pass
```

Зарезервированное слово `class` указывает на создание нового класса. Далее указываем имя класса и ставим двоеточие. Тело класса записывается с отступами относительно его определения.



PEP-8! Имя класса принято записывать в стиле TitleCase, т.е. первая буква заглавная, а остальные строчные. Если название класса состоит из нескольких слов, они записываются слитно без использования символа подчеркивания. Каждое слово с большой буквы: `class MyBestSuperData`



PEP-8! Определение класса записывается в начале файла, после импортов и констант уровня модуля. До и после класса оставляют по две пустых строки.

Экземпляры класса

Для создания экземпляра класса необходимо выполнить операцию присваивания вызвав класс. Точно так же как в примере со списком `list`.

```
class Person:
```

```
max_up = 3
```

```
p1 = Person()  
print(p1.max_up)
```

Теперь переменная `p1` является экземпляром класса `Person`. Мы можем обращаться в переменным класса из экземпляра.

Обычно в литературе приводят сравнение класса с чертежом, а его экземпляра с объектом, созданным на основе чертежа. Для Python так же будет верно сравнение класса с прототипом, а экземпляра с серийным объектом, созданным на основе прототипа. Класс — такой же объект, как и экземпляр. С ним так же можно взаимодействовать. Это не просто чертёж на бумаге.

```
print(Person.max_up)
```

Атрибуты класса и экземпляров

Переменная `max_up` в классе считается атрибутом класса. В некоторой литературе такие переменные называют свойствами класса. Также иногда используют термин поля класса. Свойства позволяют хранить значения и переходят ко всем экземплярам класса.

Рассмотрим несколько особенностей работы с атрибутами.

```
class Person:  
    max_up = 3  
  
p1 = Person()  
p2 = Person()  
print(f'{Person.max_up = }, {p1.max_up = }, {p2.max_up = }')  
p1.max_up = 12  
print(f'{Person.max_up = }, {p1.max_up = }, {p2.max_up = }')  
Person.max_up = 42  
print(f'{Person.max_up = }, {p1.max_up = }, {p2.max_up = }')
```

Изначально у класса и двух его экземпляров значения `max_up` совпадают. Экземпляры “не видят” `max_up` у себя и берут значение у класса.

После изменения свойства у экземпляра p1 мы видим новое значение у него, но старые у класса и экземпляра p2. Изменения экземпляра всегда затрагивают этот экземпляр.

Далее мы меняем значение свойства у класса. Экземпляр p2 “не видит” max_up у себя, поэтому снова обращается к классу и возвращает новое значение. Экземпляр p1 имеет собственную локальную переменную max_up, поэтому его свойство не изменилось. Изменения свойств класса затрагивают те экземпляры, которые не имеют собственных свойств.

Динамическая структура класса

Класс и экземпляр являются динамическими объектами. Мы можем добавлять атрибуты в процессе работы, а не только в момент создания класса.

```
class Person:
    max_up = 3

p1 = Person()
p2 = Person()
Person.level = 1
print(f'{Person.level = }, {p1.level = }, {p2.level = }')
p1.health = 100
print(f'{Person.health = }, {p1.health = }, {p2.health = }') #
AttributeError: type object 'Person' has no attribute 'health'
print(f'{p1.health = }, {p2.health = }') # AttributeError:
'Person' object has no attribute 'health'
print(f'{p1.health = }')
```

Добавление свойства level для класса позволяет обращаться к нему и из экземпляров.

Когда же мы добавляем свойство health для экземпляра p1, получаем ошибку AttributeError. Ни класс, ни экземпляр p2 не могут получить доступ к данному атрибуту.

Возможность динамически изменять класс может быть использована как аналог работы со словарями dict.

```
class Person:
    pass
```



```

p1 = Person()
p1.level = 1
p1.health = 100

dict_p1 = {}
dict_p1['level'] = 1
dict_p1['health'] = 100

print(f'{p1.health = }')
print(f'{dict_p1["health"] = }')

```

Если в словаре мы указываем строковой ключ в квадратных скобках, в экземпляре достаточно точечной нотации без лишних скобок и кавычек.

Конструктор экземпляра

При создании класса обычно используют функцию конструктор `__init__`.

```

class Person:
    max_up = 3

    def __init__(self):
        self.level = 1
        self.health = 100

p1 = Person()
p2 = Person()
print(f'{p1.max_up = }, {p1.level = }, {p1.health = }')
print(f'{p2.max_up = }, {p2.level = }, {p2.health = }')
print(f'{Person.max_up = }, {Person.level = }, {Person.health = }')
# AttributeError: type object 'Person' has no attribute 'level'
Person.level = 100
print(f'{Person.level = }, {p1.level = }, {p2.level = }')

```

Внутри класса создаём функцию `__init__`. Два символа подчёркивания до и после имени говорят о том, что это “магическое имя”. Подобные имена нужны для добавления новых возможностей в работе класса и его экземпляров.

Внутри функции заданы две переменные `level` и `health`. Это атрибуты экземпляров. Любой экземпляр получает заранее присвоенные значения. При этом сам класс не имеет доступа к заданным атрибутам.


Также при попытке определить свойства `level` у класса мы не меняем значения экземпляров. Это разные объекты, находящиеся в разных локальных областях видимости.

- **Параметр `self`**

Ещё раз посмотрим на код конструктора:

```
def __init__(self):  
    self.level = 1  
    self.health = 100
```

В качестве параметра указана переменная `self`. Далее мы не просто присваиваем значения переменным, а указываем `self` с точечной нотацией. В работе с классами `self` является указателем на тот экземпляр класса, к которому происходит обращение. Например для `p1` это `p1.level = 1`. Какое бы имя вы не дали экземпляру, `self` подставляет его на своё место.

 **PEP-8!** Имя `self` не является зарезервированным. Вместо него можно использовать любое. Но соглашение о написании кода требует писать `self`. Так ваш код поймут другие разработчики, а IDE верно его проанализируют.

В некоторых языках при написании кода используется запись `this.name`. При некотором допущении можно считать, что Python использует вместо `this` слово `self` с той же логикой.

- **Передача аргументов в экземпляр**

При создании экземпляра можно передать значения в конструктор и тем самым добавить свойства, характерные для конкретного экземпляра.

```
class Person:  
    max_up = 3  
  
    def __init__(self, name, race='unknown'):  
        self.name = name  
        self.race = race  
        self.level = 1  
        self.health = 100  
  
p1 = Person('Сильвана', 'Эльф')  
p2 = Person('Иван', 'Человек')  
p3 = Person('Грогу')  
print(f'{p1.name} = {p1.race}')  
print(f'{p2.name} = {p2.race}')
```

```
print(f'{p3.name = }, {p3.race = }')
```

У `__init__` определено три параметра. При этом первый параметр всегда `self` и он не учитывается при передаче аргументов. Вызывая класс ожидаются два параметра, при этом второй имеет значение по умолчанию. За исключением `self` логика такая же как и при создании обычной функции. Все изученные в теме функции знания применимы и к функциям класса.

Методы класса

Функция внутри класса называется методом. Мы уже рассмотрели магический метод `__init__`, который вызывается при создании экземпляра класса. Помимо этого можно создавать любые методы внутри класса и обращаться к ним из экземпляра через точечную нотацию. Различие между обращением к свойству и к методу - круглые скобки после имени.

```
class Person:
    max_up = 3

    def __init__(self, name, race='unknown'):
        self.name = name
        self.race = race
        self.level = 1
        self.health = 100

    def level_up(self):
        self.level += 1

p1 = Person('Сильвана', 'Эльф')
p2 = Person('Иван', 'Человек')
p3 = Person('Грогу')
print(f'{p1.level = }, {p2.level = }, {p3.level = }')
p3.level_up()
p1.level_up()
p3.level_up()
print(f'{p1.level = }, {p2.level = }, {p3.level = }')
```

Метод `level_up` берёт значение `level` у экземпляра, который вызвал этот метод и увеличивает на единицу. Работа метода не затрагивает другие экземпляры.



PEP-8! Между методами класса оставляется по одной пустой строке до и после. Как вы помните в модуле до и после функции оставляют по две пустые строки.

При желании можно передавать в метод аргументы. И так как в Python всё объект, можно передать даже экземпляр класса.

```
class Person:
    max_up = 3

    def __init__(self, name, race='unknown'):
        self.name = name
        self.race = race
        self.level = 1
        self.health = 100

    def level_up(self):
        self.level += 1

    def change_health(self, other, quantity):
        self.health += quantity
        other.health -= quantity

p1 = Person('Сильвана', 'Эльф')
p2 = Person('Иван', 'Человек')
p3 = Person('Грогу')
print(f'{p1.health = }, {p2.health = }, {p3.health = }')
p1.change_health(p2, 10)
print(f'{p1.health = }, {p2.health = }, {p3.health = }')
```

Метод `change_health` принимает ещё один экземпляр и количество здоровья. Атрибут надо изменить и у экземпляра, вызвавшего метод и у второго, переданного экземпляра.



Внимание! Чаще всего для указания на другой экземпляр того же класса используют параметр `other` в имени метода. Соответственно записи `other.name` аналогичны `self.name`, но изменяют другой, переданный экземпляр класса.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class User:
    count = []

    def __init__(self, name, phone):
        self.name = name
        self.phone = phone

u1 = User('One', '123-45-67')
u2 = User('NoOne', '76-54-321')

u1.count.append(42)
u1.count.append(73)

u2.counter = 256
u2.count.append(u2.counter)
u2.count.append(u1.count[-1])

print(f'{u1.name = }, {u1.phone = }, {u1.count = }')
print(f'{u2.name = }, {u2.phone = }, {u2.count = }')
```

2. Инкапсуляция

В ряде языков программирования под инкапсуляцией понимается сокрытие части свойств и методов класса от доступа извне класса. Как вы понимаете пользователи программы не пишут код. Они используют графический или консольный интерфейс. Или даже пользуются программой даже не догадываясь об этом, к примеру получая данные от программы-сервера во время сёрфинга в интернете. Следовательно инкапсуляция нужна одним разработчикам для сокрытия от других разработчиков и самих себя.

Python исповедует принципы разумного программирования. Если один разработчик решает внести изменения в код другого, он понимает что делает. В результате в Python нет строгой инкапсуляции как в ряде других языков. Часть инкапсуляции прописана как соглашения. И мы сталкивались с ними, когда разбирали модули и импорт переменных и функций из модулей.

Модификаторы доступа

В классическом ООП выделяют следующие модификаторы доступа:

- `public` — публичный доступ, т.е. возможность обратиться к свойству или методу из любого другого класса и экземпляра
- `protected` — защищённый доступ, позволяющий обращаться к свойствам и методам из класса и из классов наследников.
- `private` — приватный доступ, т.е. отсутствие возможности обратиться к свойству или методу из другого класса или экземпляра.

В Python по умолчанию все свойства и методы публичные. Однако существуют соглашения по стилю имён, которые делают атрибуты защищёнными/приватным. Речь в очередной раз пойдёт о символе подчёркивания.

Одно подчёркивание в начале

Одиночный символ подчёркивания в начале имени свойства или метода говорит о том, что он защищён. Мы просим других разработчиков не менять значения защищённых свойств и не вызывать защищённые методы.

```
class Person:
    max_up = 3
    _max_level = 80

    def __init__(self, name, race='unknown', speed=100):
        self.name = name
        self.race = race
        self.level = 1
        self.health = 100
        self._speed = speed

    def _check_level(self):
        return self.level < self._max_level

    def level_up(self):
        if self._check_level():
            self.level += 1
```

```

def change_health(self, other, quantity):
    self.health += quantity
    other.health -= quantity

p1 = Person('Сильвана', 'Эльф', 120)
p2 = Person('Иван', 'Человек')
p3 = Person('Грогу', speed=60)
print(f'{p1._max_level = }')
print(f'{p2._speed = }')
p2._speed = 150
print(f'{p2._speed = }')

p3.level_up()
print(f'{p3.level = }')
p3.level = 80
p3.level_up()
print(f'{p3.level = }')

```

Переменная уровня класса `_max_level` и переменная уровня экземпляра `_speed` говорят другим разработчикам, что они защищены. Так мы просим их не использовать. Однако мы сможем обратиться к ним через точечную нотацию. И даже изменить, если очень захотим. Скорее всего IDE будет указывать на неверные действия с подобными переменными.

Аналогично метод `_check_level` говорит о том, что он защищён. Метод нужен классу для проверки достижения максимального уровня персонажа и не должен использоваться напрямую вне класса. А вот его вызов из метода `level_up` считается нормальным поведением.

Два подчёркивания в начале

Двойное подчёркивание делает свойство или метод приватным. При этом срабатывает механизм защиты имени за пределами класса.



Важно! Переменная с двумя подчёркиваниями в начале не может иметь более одного подчёркивания в конце имени. Двойное подчёркивание до и после имени — магическая переменная Python. Подобно `__init__` такие имена зарезервированы для особых действий.

```
class Person:
```

```

__max_up = 3
__max_level = 80

def __init__(self, name, race='unknown', speed=100):
    self.name = name
    self.race = race
    self.level = 1
    self.health = 100
    self._speed = speed
    self.up = 3

def _check_level(self):
    return self.level < self._max_level

def level_up(self):
    if self._check_level():
        self.level += 1

def change_health(self, other, quantity):
    self.health += quantity
    other.health -= quantity

def add_up(self):
    self.up += 1
    self.up = min(self.up, self.__max_up)

p1 = Person('Сильвана', 'Эльф', 120)
print(f'{p1.up = }')
p1.up = 1
print(f'{p1.up = }')
for _ in range(5):
    p1.add_up()
    print(f'{p1.up = }')

print(p1.__max_up)      # AttributeError: 'Person' object has no
                        # attribute '__max_up'
print(Person.__max_up)  # AttributeError: type object 'Person'
                        # has no attribute '__max_up'

```

Переменная `__max_up` доступна внутри класса и его экземпляров. Мы используем её для увеличения количества жизней персонажа в методе `add_up`. Никаких проблем с доступом нет.

Когда же пытаемся обратиться к свойству напрямую, получаем ошибку доступа к атрибуту. Аналогичные ошибки будут и при обращении к методу, начинающемуся с двух подчёркиваний.

Доступ к приватным переменным

Приватная переменная `__max_up` не исчезает за пределами класса. Срабатывает механизм модификации имени. В общем случае он превращает переменную `__name` в переменную `_classname__name`.

```
class Person:
    __max_up = 3
    ...

print(f'{p1._Person__max_up = }')
```

В нашем примере переменная `__max_up` превратилась в `_Person__max_up`. Обратите внимание на заглавную `P` в имени, ведь Python является регистрозависимым языком.

Важно! У вас должны быть особые обстоятельства чтобы обращаться и тем более изменять значения переменных, которые начинаются с двух подчёркиваний за пределами их класса. Скорее всего есть другой способ решить вашу задачу.

Задание

Перед вами несколько строк кода. Какие ошибки и недочёты есть в коде. У вас 3 минуты.

```
class User:

    def __init__(self, name, phone, password):
        self.__name__ = name
        self._phone = phone
        self.__password = password

u1 = User('One', '123-45-67', 'qwerty')

print(f'{u1.__name__ = }, {u1._phone = }, {u1._User__password = }')
```

3. Наследование

В Python все объекты являются наследниками класса object. В начале лекции мы рассмотрели пример создания класса:

```
class Person:
    pass
```

Представленная запись создания класса является упрощённой. На самом деле класс Person наследуется от класса object. Т.е. object — родительский класс для Person, а Person — дочерний класс для object.

```
class Person(object):
    pass
```



Важно! Наследование от object принято опускать при создании класса.

Создадим класс героя на основе класса персонажа.

```
class Person:
    __max_up = 3
    _max_level = 80

    def __init__(self, name, race='unknown', speed=100):
        self.name = name
        self.race = race
        self.level = 1
        self.health = 100
        self._speed = speed
        self.up = 3

    def _check_level(self):
        return self.level < self._max_level

    def level_up(self):
        if self._check_level():
            self.level += 1

    def change_health(self, other, quantity):
        self.health += quantity
        other.health -= quantity
```

```

def add_up(self):
    self.up += 1
    self.up = min(self.up, self.__max_up)

class Hero(Person):
    def __init__(self, power, *args, **kwargs):
        self.power = power
        super().__init__(*args, **kwargs)

p1 = Hero('archery', 'Сильвана', 'Эльф', 120)
print(f'{p1.name = }, {p1.up = }, {p1.power = }')

```

Класс Person мы не изменяли. Он перенесён из главы про инкапсуляцию. Далее мы создаём класс Hero и указываем в скобках класс Person. Герой - дочерний класс для персонажа. Мы хотим добавить герою свойство power и прописываем его в методе инициализации. Далее вызываем метод super().__init__, т.е. метод инициализации родительского класса. Без такого вызова не будут созданы атрибуты родительского класса.

Теперь при создании экземпляра класса Hero мы вначале передаём его аргументы, а далее аргументы родительского класса Person.

Переопределение методов

При наследовании мы можем использовать в дочернем классе все общедоступные свойства и методы родительского класса. Кроме того можно создать свои. И если имена будут совпадать, произойдёт переопределение. Будут браться значения дочернего класса.

```

class Person:
    ...

class Hero(Person):
    def __init__(self, power, *args, **kwargs):
        self.power = power
        super().__init__(*args, **kwargs)

    def change_health(self, other, quantity):

```

```

        self.health += quantity * 2
        other.health -= quantity * 2

    def add_many_up(self):
        self.up += 1
        self.up = min(self.up, self._Person__max_up * 2)

p1 = Hero('archery', 'Сильвана', 'Эльф', 120)
p2 = Person('Маг', 'Троль')

print(f'{p1.health = }, {p2.health = }')
p1.change_health(p2, 10)
print(f'{p1.health = }, {p2.health = }')
p2.change_health(p1, 10)
print(f'{p1.health = }, {p2.health = }')

p1.add_many_up()
print(f'{p1.up = }')

```

В примере создан метод `change_health` с дополнительным множителем. Он срабатывает у героя. Но при вызове метода у экземпляра класса `Person` срабатывает старый метод.

В методе `add_many_ups` для обхода инкапсуляции используем запись `self._Person__max_up`. Экземпляр обращается к приватному атрибуту родительского класса, напрямую указав его.

Множественное наследование

Python поддерживает множественное наследование. Класс может быть наследником сразу двух и более классов. В некоторых языках множественное наследование недоступно по причине усложнения кода. Например наследуя класс существо от классов птицы и рыбы позволит создать летающую рыбу или плавающую птицу?

Рассмотрим вариант множественного наследования.

```

class Person:
    __max_up = 3
    __max_level = 80

    def __init__(self, name, race='unknown', speed=100):

```

```

        self.name = name
        self.race = race
        self.level = 1
        self.health = 100
        self._speed = speed
        self.up = 3

    def _check_level(self):
        return self.level < self._max_level

    def level_up(self):
        if self._check_level():
            self.level += 1

    def change_health(self, other, quantity):
        self.health += quantity
        other.health -= quantity

    def add_up(self):
        self.up += 1
        self.up = min(self.up, self.__max_up)

class Address:
    def __init__(self, country, city, street):
        self.country = country or ''
        self.city = city or ''
        self.street = street or ''

    def say_address(self):
        return f'Адрес героя: {self.country}, {self.city}, {self.street}'

class Weapon:
    def __init__(self, left_hand, right_hand):
        self.left_hand = left_hand or 'Клинок'
        self.right_hand = right_hand or 'Лук'

class Hero(Person, Address, Weapon):
    def __init__(self, power, name=None, race=None, speed=None,
                  country=None, city=None, street=None,
                  left_hand=None, right_hand=None):
        self.power = power
        Person.__init__(self, name, race, speed)
        Address.__init__(self, country, city, street)

```

```

        Weapon.__init__(self, left_hand, right_hand)

    def change_health(self, other, quantity):
        self.health += quantity * 2
        other.health -= quantity * 2

    def add_many_ups(self):
        self.up += 1
        self.up = min(self.up, self._Person__max_up * 2)

p1 = Hero('archery', 'Сильвана', 'Эльф', 120,
          country='Эльфляндия', street='Ночного эльфа',
          left_hand='Стрела')

print(f'{p1.say_address()}')
print(f'{p1.right_hand = }, {p1.left_hand = }')

```

Мы создали классы Address и Weapon. Добавив их к нашему герою, получаем сочетание атрибутов и методов всех перечисленных классов. Обратите внимание на то как происходит инициализация родительских классов внутри Hero __init__. Прописали все параметры из родительских классов в инициализации класса Hero. Далее вручную распределяем аргументы между методами __init__ каждого из родительских классов. Подобный приём не лучшая практика. При изменении параметров у родительских классов, дочерние могут перестать работать. При простых реализациях наследования достаточно функции super(). Обычно не стоит усложнять код до того состояния, когда внутренние механизмы не справляются с наследованием.

MRO

Аббревиатура MRO — method resolution order переводится как “порядок разрешения методов”. Относится этот порядок не только к методам, но и ко всем атрибутам класса. Это внутренний механизм, определяющий порядок наследования.

Забегая вперёд, иногда механизм не справляется с задачей. И чаще всего это говорит о сложности кода и неверной логики построения наследования. Т.е. нерабочий механизм наследования намекает разработчику на проблемы в его коде. Рассмотрим работу MRO на нескольких простых классах.

```

class A:
    def __init__(self):

```

```

        print('Init class A')
        self.data_a = 'A'

class B:
    def __init__(self):
        print('Init class B')
        self.data_b = 'B'

class C:
    def __init__(self):
        print('Init class C')
        self.data_c = 'C'

class D:
    def __init__(self):
        print('Init class D')
        self.data_d = 'D'

class X1(A, C):
    def __init__(self):
        print('Init class X1')
        super().__init__()

class X2(B, D):
    def __init__(self):
        print('Init class X2')
        super().__init__()

class X3(A, D):
    def __init__(self):
        print('Init class X3')
        super().__init__()

class Z(X1, X2, X3):
    def __init__(self):
        print('Init class Z')
        super().__init__()

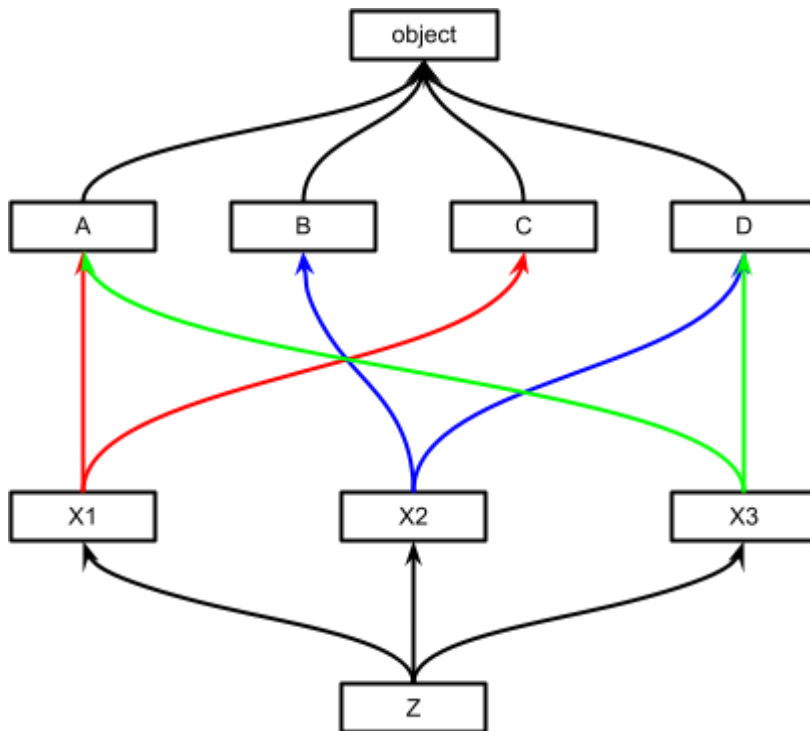
print(*Z.mro(), sep='\n')

```

1. Четыре класса A, B, C, D не имеют родительского класса. Точнее они наследуются от прародителя object. У каждого из классов есть по параметру.
2. Далее три класса X имеют по два родительских класса.
3. В финале класс Z наследуется от трёх классов X.

У каждого класса добавили текстовый вывод при вызове методу `__init__`. Также обратите внимание на наличие функции `super`, которая вызывает инициализацию родительского класса.

На схеме наследование будет выглядеть так:



У каждого класса есть метод `mro`, который вычисляет порядок наследования. Он отвечает за инициализацию каждого класса один раз в порядке слева направо и по старшинству, т.е. родитель не может быть инициализирован раньше дочернего класса.. Подробнее про то как работает “монотонная линеаризация суперкласса” можно прочитать по [ссылке](#).

Разберём результат работы `mro` с нашим классом Z.

- В первую очередь отработывает инициализация самого класса.
- Далее начинаем двигаться слева направо по списку родительских классов: X1, X2
- Следующим будет класс B. Почему он, а не X3? Класс B является родительским только для класса X2. Так мы не нарушаем порядок слева направо и старшинство.
- Следующим инициализируется X3, последний из родительских классов у Z.

- Далее идёт инициализация класса А. Он родитель для Х1 и Х3. Следовательно его инициализация была невозможна раньше дочерних классов.
- Классы С и D инициализируются последними, они правее А, В и С в списке родительских классов у “иксов”.
- Класс object всегда инициализируется в последнюю очередь.

Поиск аргументов и методов в экземпляре класса Z будет происходить в порядке, представленном методом mro.

Добавим несколько строк кода и посмотрим на результат:

```
z = Z()
print(f'{z.data_b = }')
print(f'{z.data_a = }')    # AttributeError: 'Z' object has no
attribute 'data_a'
```

Вызов метода `__init__` остановился на классе В. Мы не дописали ему вызов `super`, считая что он и так не имеет наследников. В результате аргумент `data_a` не был создан в экземпляре класса z. Попробуем описать классу А дополнительный метод.

```
class A:
    def __init__(self):
        print('Init class A')
        self.data_a = 'A'

    def say(self):
        print('Текст')
        print(self.data_b)
    ...

z = Z()
z.say()
```

Вызов метода `say` из класса А отработал без ошибок. Мы нашли его двигаясь по цепочке линейаризации. При этом метод даже смог обратиться к свойству другого класса. Связано это с тем, что мы работаем из экземпляра класса Z и он собрал в себя аргументы и методы наследуемых классов.



Важно! Не стоит из родительских классов обращаться к аргументам и методам дочерних классов или классов того же уровня наследования.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class A:
    name = 'A'

    def call(self):
        print(f'I am {self.name}')

class B:
    name = 'B'

    def call(self):
        print(f'I am {self.name}')

class C:
    name = 'C'

    def call(self):
        print(f'I am {self.name}')

class D(C, A):
    pass

class E(D, B):
    pass

e = E()
e.call()
```

4. Полиморфизм

С полиморфизмом мы уже сталкивались раньше. Например сложение чисел возвращает их сумму, а сложение строк возвращает новую строку состоящую из двух исходных. Одинаковые действия приводят к разному, но ожидаемому результату.

Полиморфизм был ранее в этой лекции:

```
class Person:
    ...

    def change_health(self, other, quantity):
        self.health += quantity
        other.health -= quantity
    ...

class Hero(Person):
    def __init__(self, power, *args, **kwargs):
        self.power = power
        super().__init__(*args, **kwargs)

    def change_health(self, other, quantity):
        self.health += quantity * 2
        other.health -= quantity * 2
    ...

p1 = Hero('archery', 'Сильвана', 'Эльф', 120)
p2 = Person('Маг', 'Тролль')
print(f'{p1.health = }, {p2.health = }')
p1.change_health(p2, 10)
print(f'{p1.health = }, {p2.health = }')
p2.change_health(p1, 10)
print(f'{p1.health = }, {p2.health = }')
```

Один и тот же метод `change_health` по разному меняет параметр `health` у обычного персонажа и у героя.

Рассмотрим ещё один вариант полиморфизма.

```
path_1 = '/home/user'
path_2 = '/my_project/workdir'
result = path_1 / path_2    # TypeError: unsupported operand
                             type(s) for /: 'str' and 'str'
```

Python не поддерживает деление строк. Но мы уже сталкивались с тем как класс `Path` из модуля `pathlib` создавал новый путь используя символ деления. Реализовать подобный полиморфизм можно например так.

```
class DivStr(str):
    def __init__(self, obj):
        self.obj = str(obj)
```

```

def __truediv__(self, other):
    first = self.obj.endswith('/')
    start = self.obj

    if isinstance(other, str):
        second = other.startswith('/')
        finish = other
    elif isinstance(other, DivStr):
        second = other.obj.startswith('/')
        finish = other.obj
    else:
        second = str(other).startswith('/')
        finish = str(other)

    if first and second:
        return DivStr(start[:-1] + finish)
    if (first and not second) or (not first and second):
        return DivStr(start + finish)
    if not first and not second:
        return DivStr(start + '/' + finish)

path_1 = DivStr('/home/user/')
path_2 = DivStr('/my_project/workdir')
result = path_1 / path_2
print(f'{result} = ', {type(result)})
print(f'{result} / "text" = ')
print(f'{result} / 42 = ')
print(f'{result} * 3 = ')

```

Создаём класс DivStr как наследник класса str. При инициализации определяем аргумент obj, который является обычной строкой. Вся магия деления строк будет спрятана в магическом методе __truediv__ который срабатывает при делении экземпляра класса DivStr на другой такой же экземпляр или на обычную строку str:

1. Первым делом определяем заканчивается ли первая часть на символ /, а саму строку сохраняем в переменной start.
2. Далее проверяем начинается ли на символ / и сохраняем вторую половинку в finish
 - a. Если вторая половинка строка, работаем с объектом other как со строкой.
 - b. Если вторая половинка экземпляр класса DivStr, работаем со свойством obj.
 - c. Если оба варианта неверны, пробуем привести объект к строковому виду.

3. В зависимости от того заканчивается или нет первая часть на символ / и начинается или нет вторая часть на символ / соединяем объекты и возвращаем новый экземпляр DivStr.

Используя полиморфизм и переопределение метода мы смогли “разделить” два экземпляра DivStr. Также мы можем “делить” на строки и даже на числа, ведь числа имеют строковое представление. А так как мы наследовались от str, объект поддерживает и привычные операции со строками.

Подробнее про переопределение магических методов поговорим на следующем занятии. Эта тема заслуживает отдельного внимания.

Вывод

На этой лекции мы:

1. Разобрались с объектно-ориентированным программированием в Python.
2. Изучили особенности инкапсуляции в языке
3. Узнали о наследовании и механизме разрешения множественного наследования.
4. Разобрались с полиморфизмом объектов.

Краткий анонс следующей лекции

1. Разберёмся с созданием и удалением классов
2. Узнаем о документировании классов
3. Изучим способы представления экземпляров
4. Узнаем о возможностях переопределения математических операций
5. Разберёмся со сравнением экземпляров
6. Узнаем об обработке атрибутов

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий и попробуйте перенести переменные и функции в класс. Убедитесь, что созданный вами класс позволяет верно решать поставленные задачи.

Оглавление

На этой лекции мы	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции	
1. Создание и удаление	4
Шаблон Одиночка, Singleton	7
Задание	9
2. Строка документации	10
Хранение документации в <code>__doc__</code>	11
3. Представления экземпляра	12
Представление для пользователя, <code>__str__</code>	13
4. Математика и логика в классах	17
Сдвиг вправо, <code>__rshift__</code>	19
Right методы	20
Умножение текста на “продвинутый текст” методом <code>__rmul__</code>	20
In place методы	21
Вычисление процентов вместо нахождения остатка от деления, <code>__imod__</code>	21
Задание	22
5. Сравнение экземпляров класса	23
Сравнение на идентичность, <code>__eq__</code>	23
Сравнение на больше и меньше	25
Неизменяемые экземпляры, хеширование дандер <code>__hash__</code>	26
Простейшая реализация хэша	28
Задание	29
6. Обработка атрибутов	29
Получение значения атрибута, <code>__getattr__</code>	30

Присвоение атрибуту значения, <code>__setattr__</code>	31
Обращение к несуществующему атрибуту, <code>__getattr__</code>	32
Удаление атрибута, <code>__delattr__</code>	33

На этой лекции мы

1. Разберёмся с созданием и удалением классов
2. Узнаем о документировании классов
3. Изучим способы представления экземпляров
4. Узнаем о возможностях переопределения математических операций
5. Разберёмся со сравнением экземпляров
6. Узнаем об обработке атрибутов

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с объектно-ориентированным программированием в Python.
2. Изучили особенности инкапсуляции в языке
3. Узнали о наследовании и механизме разрешения множественного наследования.
4. Разобрались с полиморфизмом объектов.

Термины лекции

- **Дандер** — имя переменной, начинающейся и заканчивающейся двумя подчеркиваниями. В Python такие переменные используются для создания специальных свойств и методов объекта, влияющих на его поведение.

Подробный текст лекции

1. Создание и удаление

Разберём процессы, которые происходят при создании и удалении. Python позволяет контролировать создание класса и экземпляра класса, а также удаление экземпляра. Забегая вперёд, задачи контроля создания класса и удаления экземпляра более высокого уровня. В обычной практике они используются редко. Но общие знания лишними не будут для понимания классов Python в целом.

Создание экземпляра класса, `__init__`

За создание экземпляра класса отвечает дандер метод `__init__`. С ним вы уже знакомы по прошлому занятию. Всё дело в том, что это самый частый дандер метод в ООП Python

```
class User:
    def __init__(self, name: str, equipment: list = None):
        self.name = name
        self.equipment = equipment if equipment is not None else []

        self.life = 3
        # принтим только в учебных целях, а не для реальных задач
        print(f'Создал {self} со свойствами:\n'
              f'{self.name = },\t{self.equipment = },\t{self.life'
              = }')

print('Создаём первый раз')
u_1 = User('Спенглер')
print('Создаём второй раз')
u_2 = User('Венкман', ['протонный ускоритель', 'ловушка'])
print('Создаём третий раз')
u_3 = User(equipment=['ловушка', 'прибор ночного видения'],
           name='Стэнц')
```

Класс User дважды вызывается для создания экземпляров u_1, u_2 и u_3. Вызов класса - это указание его имени с круглыми скобками после. Такой вызов автоматически перенаправляется в метод `__init__`. По сути мы вызвали его, следовательно возможность передавать аргументы при создании экземпляра прописывается параметрами “инит”.

В нашем примере name является обязательным параметром, а equipment - не обязателен. При этом значения можно передавать как позиционно, так и по ключу. В этом плане методы классов работают аналогично обычным функциям.

Внутри метода каждый аргумент присваивается переменной экземпляра. Она начинается с `self` — названия первого параметра метода. При этом для “оборудования” делается проверка на None. Если список не передан, создаётся новый для каждого экземпляра. Кроме того свойство `life` получает значение 3. Подобный подход является предпочтительным. У каждого экземпляра будет своё свойство `life` изменяющееся внутри и не зависящее от других экземпляров и от самого класса User.

Вывод на печать позволяет отследить порядок выполнения действий. Пока не отработает инициализация экземпляра, дальше код не выполняется.



Важно! Постарайтесь разобраться в работе дандер метода `__init__`. Он встречается чаще других и определяет структуру экземпляров. В подавляющем большинстве (если речь не идёт о метапрограммировании) внутри метода прописываются все свойства экземпляра.

Контроль создания класса через `__new__`

Метод `__new__` срабатывает раньше `__init__` и определяет что должен вернуть класс в качестве себя - класса. Рассмотрим вначале общий пример.

```
class User:
    def __init__(self, name: str):
        self.name = name
        print(f'Создал {self.name = }')

    def __new__(cls, *args, **kwargs):
        instance = super().__new__(cls)
        print(f'Создал класс {cls}')
        return instance

print('Создаём первый раз')
u_1 = User('Спенглер')
```

```
print('Создаём второй раз')
u_2 = User('Венкман')
print('Создаём третий раз')
u_3 = User(name='Стэнц')
```

Метод `__new__` принимает в качестве первого параметра сам себя. Обычно используют слово `cls` — сокращение от `class`. Так понятно, что мы работаем с классом, а не с его экземпляром. Параметры `*args`, `**kwargs` нужны для правильной работы метода `__init__` и попадания в него любых аргументов.

Внутри `__new__` необходимо вызвать аналогичный родительский метод. Он возвращает класс, который можно модифицировать прежде чем вернуть из метода.

Расширение неизменяемых классов

Один из вариантов использования дандер `__new__` — расширение функциональности уже имеющихся неизменяемых классов Python. Например мы хотим использовать переменную целого типа, которая дополнительно хранит присвоенное числу имя.

```
class NamedInt(int):
    def __new__(cls, value, name):
        instance = super().__new__(cls, value)
        instance.name = name
        print(f'Создал класс {cls}')
        return instance

print('Создаём первый раз')
a = NamedInt(42, 'Главный ответ жизни, Вселенной и вообще')
print('Создаём второй раз')
b = NamedInt(73, 'Лучшее просто число')
print(f'{a = }\t{a.name = }\t{type(a) = }')
print(f'{b = }\t{b.name = }\t{type(b) = }')
c = a + b
print(f'{c = }\t{type(c) = }')
```

Параметр `value` нужен для передачи значения в родительский класс `int`. Далее к целому числу добавляется параметр `name` с переданным значением. После создания объекта он возвращается для присваивания переменной.

Обратите внимание, что наш класс унаследовал всё, что умеет класс `int`. Мы смогли сложить два числа и получить обычное целое число без свойства `name`.

Шаблон Одиночка, Singleton

Ещё один вариант использования дандре `__new__` — паттерн Singleton. Он позволяет создавать лишь один экземпляр класса. Вторая и последующие попытки будут возвращать ранее созданный экземпляр.



Внимание! Это не единственный вариант создания паттерна Одиночка в Python, а версия через `__new__`

```
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    def __init__(self, name: str):
        self.name = name

a = Singleton('Первый')
print(f'{a.name = }')
b = Singleton('Второй')
print(f'{a is b = }')
print(f'{a.name = }\n{b.name = }')
```

Защищенная переменная `_instance` хранит `None` и при создании первого экземпляра получает на него ссылку. Благодаря `*args, **kwargs` в методе `__new__` аргумент “Первый” проваливается в метод `__init__` и попадает в параметр `name`. При создании экземпляра второй раз возвращается его первая версия и у неё заменяется свойство `name`. Переменные `a` и `b` ссылаются на один и тот же класс, а следовательно их свойство `name` общее.

Удаление экземпляра класса, `__del__`

Дандер метод `__del__` также редко используется в обычной практике ООП. Он отвечает за действия при удалении экземпляра класса. Если быть более точным, метод срабатывает при достижении нуля счётчиком ссылок на объект, но перед его удалением из памяти сборщиком мусора. Рассмотрим простой пример.

```
class User:
    def __init__(self, name: str):
        self.name = name
        print(f'Создал {self.name = }')

    def __del__(self):
        print(f'Удаление экземпляра {self.name}')

u_1 = User('Спенглер')
u_2 = User('Венкман')
```

После создания двух экземпляров программа завершилась. Начался процесс освобождения памяти и перед удалением экземпляров был вызван их метод `__del__`. Обычно метод `__del__` используют для явного освобождения других ресурсов. Например экземпляр использует соединения с базой данных и перед удалением его необходимо корректно завершить.



Важно! Python в любом случае удалит бы экземпляры и освободил память, даже если дандер `__del__` не прописан.

Команда del

Зарезервированное слово `del` не удаляет объекты в Python. Оно разрывает связь между переменной и объектом, на который переменная указывает. После строки с `del` к переменной нельзя обратиться, а у объекта на единицу уменьшается счётчик ссылок.

```
import sys

class User:
    def __init__(self, name: str):
        self.name = name
```

```

print(f'Создал {self.name = }')

def __del__(self):
    print(f'Удаление экземпляра {self.name}')

u_1 = User('Спенглер')
print(sys.getrefcount(u_1))
u_2 = u_1
print(sys.getrefcount(u_1), sys.getrefcount(u_2))
del u_1
print(sys.getrefcount(u_2))
print('Завершение работы')
```

В момент первого вызова метода `getrefcount` имеем одно значение счётчика ссылок. Когда переменная `u_2` получает ссылку на тот же объект, счётчик ссылок увеличивается на единицу. Команда `del` уменьшает счётчик на единицу. А удаление объекта происходит после завершения кода.



Важно! Счётчик ссылок возвращает значение больше ожидаемого, так как сама функция создаёт дополнительную ссылку на объект при вызове.

Если в коде дописать `del u_2` в предпоследней строке, удаление объекта произойдёт раньше завершения работы программы. Счётчик ссылок дошёл до нуля и сборщик мусора освободил память.

Задание

Перед вами несколько строк кода. Напишите что делает класс, не запуская код. У вас 3 минуты.

```

class Count:
    _count = 0
    _last = None

    def __new__(cls, *args, **kwargs):
        if cls._count < 3:
            cls._last = super().__new__(cls)
            cls._count += 1
```

```
        return cls._last

    def __init__(self, name: str):
        self.name = name
```

2. Строка документации

Как и при создании функции, при создании классов принято оставлять документацию к нему. Для этого достаточно использовать многострочный комментарий сразу после определения класса — строки `class ClassName:`. Посмотрите на пример

```
class User:
    """A User training class for demonstrating class
    documentation.
    Shows the operation of the help(cls) and the dander method
    __doc__"""

    def __init__(self, name: str):
        """Added the name parameter."""
        self.name = name

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

u_1 = User('Спенглер')
print('Справка класса User ниже', '*' * 50)
help(User)
print('Справка экземпляра u_1 ниже', '*' * 50)
help(u_1)
```

После заголовка класса оставили пару строк описания. Также добавлена строка документации для метода `__init__`. Далее вызываем справку для класс `User` и для его экземпляра. С первого взгляда они похожи и выводят куда больше информации. Помимо оставленной документации `help` собирает информацию о методах класса. Выводит первую строку метода и строку документации метода, если она задана многострочным комментарием.

Отличие справки для класса и экземпляра лишь в первой строке. Сравните:

Help on class User in module `__main__`:

Help on User in module `__main__` object:

Хранение документации в `__doc__`

Любая многострочная строка после заголовка класса и метода автоматически сохраняется в дандер переменную `__doc__`. Помимо вызова справки через функцию `help` можно прочитать отдельный многострочник напрямую обратившись к переменной.

```
class User:
    """A User training class for demonstrating class
    documentation.
    Shows the operation of the help(cls) and the dander method
    __doc__"""

    def __init__(self, name: str):
        """Added the name parameter."""
        self.name = name
        print(f'Создал {self.name = }')

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

u_1 = User('Спенглер')
print(f'Документация класса: {User.__doc__ = }')
print(f'Документация экземпляра: {u_1.__doc__ = }')
print(f'Документация метода: {u_1.simple_method.__doc__ = }')
```

Как и в случае с `help` обращение через класс или через экземпляр не даёт разницы.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class MyClass:
    A = 42
```



```

"""About class"""

def __init__(self, a, b):
    """self.__doc__ = None"""
    self.a = a
    self.b = b

def method(self):
    """Documentation"""
    self.__doc__ = None

help(MyClass)

```

3. Представления экземпляра

При работе с классами, а точнее с их экземплярами бывает необходимо вывести их содержимое в консоль. С этим отлично справляется функция `print`, но есть одно но. Попробуем “запринтить” класс из примера выше.

```

class User:

    def __init__(self, name: str):
        """Added the name parameter."""
        self.name = name

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

user = User('Спенглер')
print(user)

```

В результат получили строку вида `<__main__.User object at 0x000001C1B4FA6B60>`. Число в конце — адрес объекта в оперативной памяти. Он может быть разным для разных ПК и даже при разных запусках программы. Но пользы от этой информации немного. Для получения читаемого описания необходимо переопределить как минимум один из дандер методов: `__str__` или `__repr__`.

Представление для пользователя, `__str__`

Дандер метод `__str__` используется для получения удобного пользователю описания экземпляра.

```
class User:

    def __init__(self, name: str):
        """Added the name parameter."""
        self.name = name

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

    def __str__(self):
        return f'Экземпляр класса User с именем "{self.name}"'

user = User('Спенглер')
print(user)
```

Метод `__str__` обязан вернуть строку `str`. Обычно это строка содержит информацию о свойствах класса для понимания что за экземпляр перед нами. Упор делается на удобство чтения. Но и о краткости забывать не стоит.

Представление для создания экземпляра, `__repr__`

Дандер метод `__repr__` аналогичен `__str__`, но возвращает максимально близкое к созданию экземпляра класса представление.

```
class User:

    def __init__(self, name: str):
```

```

        """Added the name parameter."""
        self.name = name

    def simple_method(self):
        """Example of a simple method."""
        self.name.capitalize()

    def __repr__(self):
        return f'User({self.name})'

user = User('Спенглер')
print(user)

```

Если скопировать вывод метода `repr` и присвоить его переменной, должен получится ещё один экземпляр класса. Рассмотрим более сложный класс и его метод `__repr__`.

```

class User:

    def __init__(self, name: str, equipment: list = None):
        self.name = name
        self.equipment = equipment if equipment is not None else
[]
        self.life = 3

    def __repr__(self):
        return f'User({self.name}, {self.equipment})'

user = User('Венкман', ['протонный ускоритель', 'ловушка'])
print(user)

```

Мы снова получили строку, которую можно скопировать и создать экземпляр без внесения правок. При этом свойство `life` опущено в выводе, т.к. не влияет на создание экземпляра.

Приоритет методов

Добавим классу из примера выше метод `__str__` и посмотрим какой из них сработает.

```

class User:

```

```

def __init__(self, name: str, equipment: list = None):
    self.name = name
    self.equipment = equipment if equipment is not None else
[]
    self.life = 3

def __str__(self):
    eq = 'оборудованием: ' + ', '.join(self.equipment) if
self.equipment else 'пустыми руками'
    return f'Перед нами {self.name} с {eq}. Количество жизней
- {self.life}'

def __repr__(self):
    return f'User({self.name}, {self.equipment})'

user = User('Венкман', ['протонный ускоритель', 'ловушка'])
print(user)

```

При вызове функции print сработал метод __str__. Как же получить вывод от __repr__ при наличии двух методов? Есть несколько способов вывода на печать:

```

class User:
    ...

user = User('Венкман', ['протонный ускоритель', 'ловушка'])
print(user)
print(f'{user}')

print(repr(user))
print(f'{user = }')

```

В первых двух вариантах срабатывает дандер __str__. Далее мы явно передаём в print результат встроенной функции repr, которая обращается к одноимённому методу. Так же при использовании f-строк символ равенства выводит имя переменной слева от знака равно и repr справа от него.

Печать коллекций

Однако метод __repr__ оказывается более приоритетным, если на печать выводится не один элемент, а коллекция элементов.

```

class User:

    def __init__(self, name: str, equipment: list = None):
        self.name = name
        self.equipment = equipment if equipment is not None else []
        self.life = 3

    def __str__(self):
        eq = 'оборудованием: ' + ', '.join(self.equipment) if self.equipment else 'пустыми руками'
        return f'Перед нами {self.name} с {eq}. Количество жизней - {self.life}'

    def __repr__(self):
        return f'User({self.name}, {self.equipment})'

u_1 = User('Спенглер')
u_2 = User('Венкман', ['протонный ускоритель', 'ловушка'])
u_3 = User(equipment=['ловушка', 'прибор ночного видения'], name='Стэнц')
ghostbusters = [u_1, u_2, u_3]

print(ghostbusters)
print(f'{ghostbusters}')
print(repr(ghostbusters))
print(f'{ghostbusters = }')

print(*ghostbusters, sep='\n')

```

В приведённом примере список из трёх экземпляров при печати возвращает герг представление во всех четырёх рассмотренных способах. И только при распаковке списка через звёздочку функция print получает экземпляры напрямую и вызывает их дандер `__str__`.

Задание

Перед вами несколько строк кода. Что в нём неправильно. У вас 3 минуты.

```

class MyClass:

    def __init__(self, a, b):

```

```

self.a = a
self.b = b
self.c = a + b

def __str__(self):
    return f'MyClass(a={self.a}, b={self.b}, c={self.c})'

def __repr__(self):
    return str(self.a) + str(self.b) + str(self.c)

```

4. Математика и логика в классах

Под математикой стоит понимать переопределение дандер методов, которые позволяют производить операции сложения, вычитания, умножения и т.п. с использованием математических символов. Что касается логики, речь идёт о логических операциях “и”, “или” и т.п. над объектами. Рассмотрим возможности Python в таблице.

Операция в Python	Основной метод	Right метод	In place метод
+	<code>__add__(self, other)</code>	<code>__radd__(self, other)</code>	<code>__iadd__(self, other)</code>
-	<code>__sub__(self, other)</code>	<code>__rsub__(self, other)</code>	<code>__isub__(self, other)</code>
*	<code>__mul__(self, other)</code>	<code>__rmul__(self, other)</code>	<code>__imul__(self, other)</code>
@	<code>__matmul__(self, other)</code>	<code>__rmatmul__(self, other)</code>	<code>__imatmul__(self, other)</code>
/	<code>__truediv__(self, other)</code>	<code>__rtruediv__(self, other)</code>	<code>__itruediv__(self, other)</code>
//	<code>__floordiv__(self, other)</code>	<code>__rfloordiv__(self, other)</code>	<code>__ifloordiv__(self, other)</code>
%	<code>__mod__(self, other)</code>	<code>__rmod__(self, other)</code>	<code>__imod__(self, other)</code>
<code>divmod()</code>	<code>__divmod__(self, other)</code>	<code>__rdivmod__(self, other)</code>	<code>__idivmod__(self, other)</code>
<code>**</code> , <code>pow()</code>	<code>__pow__(self, other[, modulo])</code>	<code>__rpow__(self, other[, modulo])</code>	<code>__ipow__(self, other[, modulo])</code>
<<	<code>__lshift__(self, other)</code>	<code>__rlshift__(self, other)</code>	<code>__ilshift__(self, other)</code>
>>	<code>__rshift__(self, other)</code>	<code>__rrshift__(self, other)</code>	<code>__irshift__(self, other)</code>
&	<code>__and__(self, other)</code>	<code>__rand__(self, other)</code>	<code>__iand__(self, other)</code>

^	<code>__xor__(self, other)</code>	<code>__rxor__(self, other)</code>	<code>__ixor__(self, other)</code>
	<code>__or__(self, other)</code>	<code>__ror__(self, other)</code>	<code>__ior__(self, other)</code>

Переопределение перечисленных в таблице методов позволяет использовать указанные в первом столбце операции для вычисления результата. Рассмотрим некоторые из них на примерах.

Обычные методы

Начнём с методов из второго столбца. Если Python встречает два экземпляра класса с одним из знаков между ними, ищется соответствующий знаку дандер метод для вызова. Если метод не определён, возвращается ошибка. При этом метод должен возвращать новый экземпляр класса без изменения исходных.

Сложение через `__add__`

Создадим класс вектор и научим вектора складываться.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

a = Vector(2, 4)
b = Vector(3, 7)
c = a + b
print(f'{a = }\t{b = }\t{c = }')
```

Помимо уже привычных методов `__init__` и `__repr__` определили метод `__add__`. В предпоследней строке пытаемся сложить вектора. Без метода `__add__` получили бы ошибку вида: `TypeError: unsupported operand type(s) for +: 'Vector' and 'Vector'`.

В самом методе используем два параметра — `self` для обращения к элементам экземпляра и `other` для обращения к элементам другого объекта, стоящего справа от знака плюс. Получив значения `x`, `y` для нового вектора метод возвращает его — новый экземпляр класса `Vector`.

Сдвиг вправо, `__rshift__`

Переопределение методов не обязательно должно быть для чисел. Напишем класс, который генерирует шкаф с одеждой и выбрасывает указанное количество вещей при правом сдвиге. Не забудем, что дандер метод должен возвращать новый экземпляр.

```
from random import choices

class Closet:
    CLOTHES = ('брюки', 'рубашка', 'костюм', 'футболка',
               'перчатки', 'носки', 'туфли')

    def __init__(self, count: int, storeroom=None):
        self.count = count
        if storeroom is None:
            self.storeroom = choices(self.CLOTHES, k=count)
        else:
            self.storeroom = storeroom

    def __str__(self):
        names = ', '.join(self.storeroom)
        return f'Осталось вещей в шкафу {self.count}: \n{names}'

    def __rshift__(self, other):
        shift = self.count if other > self.count else other
        self.count -= shift
        return Closet(self.count, choices(self.storeroom,
                                           k=self.count))

storeroom = Closet(10)
print(storeroom)
```



```
for _ in range(4):
    storeroom = storeroom >> 3
    print(storeroom)
```

Константа CLOTHES хранит доступный список одежды. Из него будем выбирать count предметов. Внутри __rshift__ сделали проверку на оставшееся количество вещей, чтобы не выбросить больше, чем уже имеется. Метод возвращает новый экземпляр, где count уменьшился на сдвиг, а второй аргумент содержит выборку из уже лежащих в шкафу вещей.

Создав экземпляр на 10 вещей и последовательно удаляем по три предмета в цикле. Но уйти в минус не удаётся, обрабатывает наша защита.

Right методы

Right методы срабатывают в том случае, если у левого аргумента в выражении метод не был найден. Например при записи `x + y` вначале производится поиска дандер метода `x.__add__`. Если он не найден, вызываем `y.__radd__`.

Умножение текста на “продвинутый текст” методом __rmul__

Создадим класс на основе `str` с методом `__rmul__`. Если слева оказывается обычная строка, будем между словами добавлять текст из “продвинутой строки”, перемножим их.

```
class StrPro(str):
    def __new__(cls, *args, **kwargs):
        instance = super().__new__(cls, *args, **kwargs)
        return instance

    def __rmul__(self, other: str):
        words = other.split()
        result = self.join(words)
        return StrPro(result)
```

```

text = 'Каждый охотник желает знать где сидит фазан'
s = StrPro(' (=^.^=) ')
print(f'{text = }\n{s = }')
print(text * s)
print(s * text)  # TypeError: 'str' object cannot be interpreted
                  as an integer

```

Метод `__new__` позволили нам наследоваться от класса `str` и забрать все свойства и методы, определённые в нём. Мы добавили лишь `__rmul__` где делим строку стоящую слева от знака умножить - `other` на отдельные слова. Далее собираем новую строку с добавлением `self`- строки справа от знака умножения.

При умножении `str` на `StrPro` получаем ожидаемый результат. Если же поменять значения местами, получаем ошибку. Обычную строку можно умножить на целое число, но не другой экземпляр.

In place методы

In place методы используются при короткой записи математического символа слитно со знаком равенства: `a += b`. Такая запись подразумевает внесение изменений в исходный объект, а не возврат нового экземпляра. Возвращать надо самого себя — `self`.

Вычисление процентов вместо нахождения остатка от деления, `__imod__`

Создадим простой класс `Money`, который будет увеличивать значение на указанный процент при записи `Money %= float | int`

```

from decimal import Decimal

class Money:
    def __init__(self, value: int | float):
        self.value = Decimal(value)

```

```

def __repr__(self):
    return f'Money({self.value:.2f})'

def __imod__(self, other):
    self.value = self.value * Decimal(1 + other / 100)
    return self

```

```

m = Money(100)
print(m)
m %= 50
print(m)
m %= 100
print(m)

```

Для точности вычислений используется класс `Decimal`. Поэтому при увеличении на указанный процент используем дополнительное обёртывание правого значения в `Decimal`.



Важно! Не забывайте `return self` при работе с in place дандер методами.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```

class MyClass:

    def __init__(self, data):
        self.data = data

    def __and__(self, other):
        return MyClass(self.data + other.data)

    def __str__(self):
        return str(self.data)

a = MyClass((1, 2, 3, 4, 5))
b = MyClass((2, 4, 6, 8, 10))
print(a & b)

```

5. Сравнение экземпляров класса

Числа сравниваются по значению, строки посимвольно. Но при желании можно сравнивать любые объекты Python реализовав перечисленные ниже дандер методы.

- `__eq__` - равно, `==`
- `__ne__` - не равно, `!=`
- `__gt__` - больше, `>`
- `__ge__` - не больше, меньше или равно, `<=`
- `__lt__` - меньше, `<`
- `__le__` - не меньше, больше или равно, `>=`

Перечисленные методы попарно противоположны. Обратите внимание на приставку не в списке. Реализовав один из пары, второй Python попытается получить инвертируя значение. Не истина — это ложь, а не ложь — это истина.

При реализации метода обычно принято возвращать `True` или `False`. Если возвращается другое значение в конструкциях вида `if x == y:`, Python применит функцию `bool()` к результату для получения `True` или `False`.

Сравнение на идентичность, `__eq__`

Создадим класс треугольник, который хранит длины трёх сторон. В первом варианте не будем прописывать дандер `__eq__` и попробуем сравнить экземпляры.

```
class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __str__(self):
        return f'Треугольник со сторонами: {self.a}, {self.b}, {self.c}'

one = Triangle(3, 4, 5)
two = one
three = Triangle(3, 4, 5)
print(one == two)
print(one == three)
```

Переменные one и two равны, т.к. ссылаются на один и тот же объект в памяти. А вот треугольники one и three считаются разными хоть и имеют одинаковые длины сторон. Дело в том, что Python по умолчанию добавляет метод `__eq__` следующего вида.

```
def __eq__(self, other):  
    return self is other
```

Как вы помните `is` сравнивает адреса объектов в памяти. Следовательно проверка по умолчанию это: `True if id(self) == id(other) else False`.

А теперь напишем свою проверку на идентичность. Допустим возможность переворачивания треугольника перед сравнением. Например треугольники со сторонами 3, 4, 5 и 4, 3, 5 будем считать равными.

```
class Triangle:  
    def __init__(self, a, b, c):  
        self.a = a  
        self.b = b  
        self.c = c  
  
    def __str__(self):  
        return f'Треугольник со сторонами: {self.a}, {self.b}, {self.c}'  
  
    def __eq__(self, other):  
        first = sorted((self.a, self.b, self.c))  
        second = sorted((other.a, other.b, other.c))  
        return first == second  
  
one = Triangle(3, 4, 5)  
two = one  
three = Triangle(3, 4, 5)  
four = Triangle(4, 3, 5)  
print(f'{one == two    = }')  
print(f'{one == three = }')  
print(f'{one == four  = }')  
print(f'{one != one   = }')
```

Функция `sorted` получает кортеж из трёх сторон и возвращает их в упорядоченном виде. Сравнив оба списка поэлементно определяем равны треугольники или нет. Обратите внимание на последнюю строку. Проверка на неравенство не вызвала ошибку. Python вызвал дандер `__eq__`, а к результату применил команду `not`.

Сравнение на больше и меньше

При необходимости сравнивать объекты не только на равенство, можно реализовать дополнительные методы сравнения. Например для работы сортировки используется метод `__lt__`, проверяющий какой из пары элементов меньше.

Доработаем класс треугольника и будем сравнивать их по площади. Для вычисления площади напишем отдельный метод, использующий [формулу Герона](#).

```
from math import sqrt

class Triangle:
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

    def __str__(self):
        return f'Треугольник со сторонами: {self.a}, {self.b}, {self.c}'

    def __repr__(self):
        return f'Triangle({self.a}, {self.b}, {self.c})'

    def __eq__(self, other):
        first = sorted((self.a, self.b, self.c))
        second = sorted((other.a, other.b, other.c))
        return first == second

    def area(self):
        p = (self.a + self.b + self.c) / 2
        _area = sqrt(p * (p - self.a) * (p - self.b) * (p - self.c))
        return _area

    def __lt__(self, other):
        return self.area() < other.area()

one = Triangle(3, 4, 5)
two = Triangle(5, 5, 5)
print(f'{one} имеет площадь {one.area():.3f} y.e.2')
print(f'{two} имеет площадь {two.area():.3f} y.e.2')
print(f'{one > two} = {\n{one < two} = }')
```

```
data = [Triangle(3, 4, 5), Triangle(6, 2, 5), Triangle(4, 4, 4),
Triangle(3, 5, 3)]
result = sorted(data)
print(result)
print(', '.join(f'{item.area():.3f}' for item in result))
```

Рассмотрим получившийся код:

- дандер `__init__` и `__str__` остались без изменений
- дандер `__repr__` нужен для вывода читаемого списка экземпляров
- дандер `__eq__` также не изменился
- метод `area` вычисляет площадь треугольника по формуле Герона:
 - находим p как половину суммы длин сторон
 - вычисляем площадь как корень из произведения p на разность p с каждой из сторон
- дандер `__lt__` вызывает для каждого из сравниваемых экземпляров метод `area` и возвращает результат сравнения площадей: `True` или `False`.

В основном коде создали пару треугольников, посмотрели на их площадь и убедились, что сравнения на больше так же работает и возвращает обратное от сравнения на меньше.

Далее создали список треугольников и отсортировали их. Визуальная проверка площадей подтверждает, что треугольники были упорядочены именно на основе их сравнения.

Неизменяемые экземпляры, хеширование дандер `__hash__`

Как вы помните ключом `dict` и элементами `set` и `frozenset` могут быть только неизменяемые типы данных. А для проверки на неизменяемость используется функция `hash()`. Она должна возвращать целое число в 4 или 8 байт в зависимости от разрядности интерпретатора Python. И это число должно быть неизменным на всём протяжении работы программы.

Попробуем сложить наши треугольники из примера выше в множество не изменяя код.

```
from math import sqrt

class Triangle:
    ...
```

```
triangle_set = {Triangle(3, 4, 5), Triangle(6, 2, 5), Triangle(4,
4, 4), Triangle(3, 5, 3)}
print(triangle_set)
```

Получаем ошибку `TypeError: unhashable type: 'Triangle'` ведь дандер `__hash__` у нас не реализован. Прежде чем приступить к написанию кода попробуем закомментировать метод проверки на равенство и запустит код снова.

```
from math import sqrt

class Triangle:
    ...
    # def __eq__(self, other):
    #     first = sorted((self.a, self.b, self.c))
    #     second = sorted((other.a, other.b, other.c))
    #     return first == second
    ...

triangle_set = {Triangle(3, 4, 5), Triangle(6, 2, 5), Triangle(4,
4, 4), Triangle(3, 5, 3)}
print(triangle_set)
print(', '.join(f'{hash(item)}' for item in triangle_set))
```

Работает! Экземпляры треугольника стали хэшируемыми. Правило следующее.

- нет `__eq__`, нет `__hash__` - неизменяемый объект. Python сам реализует оба дандера
- есть `__eq__`, нет `__hash__` - изменяемый объект. Python устанавливает `__hash__ = None`
- есть `__eq__`, есть `__hash__` - неизменяемый объект реализованный разработчиком
- нет `__eq__`, есть `__hash__` - запрещённая комбинация! Разработчик допустил ошибку

Если вы хотите явно отключить поддержку хэширования, в определение класса добавляется строка `__hash__ = None`

```
class Triangle:
    __hash__ = None
    ...
```


Простейшая реализация хэша

При желании реализовать собственный метод `__hash__` рекомендуется сделать все свойства класса неизменяемыми. Внутри дандер метода возвращается результат работы функции `hash()`. На вход функция получает кортеж из всех свойств класса.

```
from math import sqrt

class Triangle:
    def __init__(self, a, b, c):
        self._a = a
        self._b = b
        self._c = c

    def __str__(self):
        return f'Треугольник со сторонами: {self._a}, {self._b}, {self._c}'

    def __repr__(self):
        return f'Triangle({self._a}, {self._b}, {self._c})'

    def __eq__(self, other):
        first = sorted((self._a, self._b, self._c))
        second = sorted((other._a, other._b, other._c))
        return first == second

    def area(self):
        p = (self._a + self._b + self._c) / 2
        _area = sqrt(p * (p - self._a) * (p - self._b) * (p - self._c))
        return _area

    def __lt__(self, other):
        return self.area() < other.area()

    def __hash__(self):
        return hash((self._a, self._b, self._c))

triangle_set = {Triangle(3, 4, 5), Triangle(6, 2, 5), Triangle(4, 4, 4), Triangle(3, 5, 3)}
print(triangle_set)
print(', '.join(f'{hash(item)}' for item in triangle_set))
```

Свойства класса получили символ подчёркивания в начале имени. Сообщаем коллегам по коду, что они защищённые и не должны изменяться.



Важно! Как вы помните это договорённость, которую можно обойти. Философия Python надеется на разумного человека, который понимает что делать, а что нет.

Сам дандер `__hash__` возвращает результат вычисления хэша для кортежа из трёх элементов — длин сторон.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class MyClass:

    def __init__(self, a, b):
        self.a = a
        self.b = b
        self.c = a + b

    def __str__(self):
        return f'MyClass(a={self.a}, b={self.b}, c={self.c})'

    def __eq__(self, other):
        return (sum((self.a, self.b)) - self.c) == (sum((other.a,
other.b)) - other.c)

x = MyClass(42, 2)
y = MyClass(73, 3)
print(x == y)
```

6. Обработка атрибутов

Python имеет четыре дандер метода, которые позволяют контролировать обращения к атрибутам экземпляра. Разберём их на простом примере класса вектор.

```

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

a = Vector(2, 4)

```

Получение значения атрибута, __getattr__

Дандер `__getattr__` вызывается при любой попытке обращения к атрибутам экземпляра.

```

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'


    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __getattr__(self, item):
        if item == 'z':
            raise AttributeError('У нас вектор на плоскости, а не в пространстве')
        return object.__getattr__(self, item)

```

```
a = Vector(2, 4)
print(a.z) # AttributeError: У нас вектор на плоскости, а не в пространстве
print(f'{a = }')
```

В параметр `item` попадает имя атрибута, к которому пытаются обратиться в виде `str`. Мы прописали проверку имён и если это третья координата `z`, вызываем ошибку `AttributeError`.

 **Важно!** Строка `return object.__getattr__(self, item)` является обязательной. Без неё может возникнуть ошибка переполнения стека.

Присвоение атрибуту значения, `__setattr__`

Дандер `__setattr__` срабатывает каждый раз, когда в коде есть операция присвоения. Слева от знака равно экземпляр со свойством - `key`, справа значение - `value`.

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __getattr__(self, item):
        if item == 'z':
            raise AttributeError('У нас вектор на плоскости, а не в пространстве')
        return object.__getattr__(self, item)

    def __setattr__(self, key, value):
        if key == 'z':
            raise AttributeError('У нас вектор на плоскости, а не в пространстве')
```

```

в пространстве')
    return object.__setattr__(self, key, value)

a = Vector(2, 4)
print(a.z)    # AttributeError: У нас вектор на плоскости, а не в
               пространстве
print(f'{a = }')
a.z = 73      # AttributeError: У нас вектор на плоскости, а не в
               пространстве
a.x = 3
print(f'{a = }')

```

Дандер `__setattr__` запрещает присваивать значение свойству. Как и в случае с `__getattr__` важная последняя строка. Она позволяет избежать рекурсии и присвоить значение свойству, которое мы не обработали ранее.

Обращение к несуществующему атрибуту, `__getattr__`

Если свойство отсутствует, в первую очередь вызывается дандер `__getattr__`. В случае возврата им ошибки `AttributeError` вызывается метод `__getattr__`. Он также может поднять ошибку. А может как-то иначе обработать запрос.

```

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __getattr__(self, item):
        if item == 'z':
            raise AttributeError('У нас вектор на плоскости, а не
в пространстве')
        return object.__getattr__(self, item)

```

```

def __setattr__(self, key, value):
    if key == 'z':
        raise AttributeError('У нас вектор на плоскости, а не
в пространстве')
    return object.__setattr__(self, key, value)

def __getattr__(self, item):
    return None

a = Vector(2, 4)
print(a.z)  # None
print(f'{a = }')

```

Мы пропасаила возврат None для любого свойства, которое не удалось найти. Метод возвращает None и для свойства z, перехватывая исключение.

Удаление атрибута, __delattr__

При попытке удалить атрибут командой del можно использовать дандер __delattr__ для изменения логики.

```

class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __repr__(self):
        return f'Vector({self.x}, {self.y})'

    def __add__(self, other):
        x = self.x + other.x
        y = self.y + other.y
        return Vector(x, y)

    def __getattr__(self, item):
        if item == 'z':
            raise AttributeError('У нас вектор на плоскости, а не
в пространстве')
        return object.__getattr__(self, item)

    def __setattr__(self, key, value):

```

```

        if key == 'z':
            raise AttributeError('У нас вектор на плоскости, а не
в пространстве')
        return object.__setattr__(self, key, value)

    def __getattr__(self, item):
        return None

    def __delattr__(self, item):
        if item in ('x', 'y'):
            setattr(self, item, 0)
        else:
            object.__delattr__(self, item)

a = Vector(2, 4)
a.s = 73
print(f'{a = }, {a.s = }')
del a.x
del a.s
print(f'{a = }, {a.s = }')

```

В нашем классе при попытке удалить `x` или `y`, значение не удаляется. Вместо этого свойству присваивается ноль.

Обратите внимание на свойство `s`. Мы смогли присвоить ему значение 73. Дандер `__setattr__` контролирует только имя `z`. При удалении свойства `z` сработала ветка `else` и свойство было удалено. Однако мы не получили ошибки, обращаясь к несуществующему свойству, сработал дандер `__getattr__`.

Функции `setattr()`, `getattr()` и `delattr()`

В примере выше мы вызвали функцию `setattr` для присвоения у объекта `self` свойству `item` значения 0. В Python есть функции, которые позволяют делать тоже самое, что и рассмотренные выше дандер методы. Разница лишь в том, что метод реагирует на событие в коде, а функцию вы вызываете в тот момент, когда вам это нужно.

- `setattr(object, name, value)` — аналог `object.name = value`
- `getattr(object, name[, default])` — аналог `object.name` or `default`
- `delattr(object, name)` — аналог `del object.name`

Вывод

На этой лекции мы:

1. Разобрались с созданием и удалением классов
2. Узнали о документировании классов
3. Изучили способы представления экземпляров
4. Узнали о возможностях переопределения математических операций
5. Разобрались со сравнением экземпляров
6. Узнали об обработке атрибутов

Краткий анонс следующей лекции

1. Разберёмся в превращении объекта в функцию
2. Изучим способы создания итераторов
3. Узнаем о создании менеджеров контекста
4. Разберемся в превращении методов в свойства
5. Изучим работу дескрипторов
6. Узнаем о способах экономии памяти

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий и попробуйте перенести переменные и функции в класс. Добавьте к ним дандер методы из лекции для решения изначальной задачи.

Оглавление

На этой лекции мы	2
Дополнительные материалы к лекции	2
Краткая выжимка, о чём говорилось в предыдущей лекции	2
Термины лекции	3
Подробный текст лекции	
1. Класс как функция	3
2. Создаём итераторы	5
3. Создаём менеджер контекста with	8
4. Декоратор @property	11
Getter	12
Setter	13
Deleter	14
Задание	15
5. Дескрипторы	16
Класс-дескриптор Range	19
Контроль имён, __set_name__	19
Контроль получения значений, __get__	20
Контроль присвоение значений, __set__	20
Контроль удаления атрибутов, __delete__	20
Класс Student	20
6. Экономим память	21
Краткий анонс следующей лекции	24

На этой лекции мы

1. Разберёмся с созданием и удалением классов

2. Узнаем о документировании классов
3. Изучим способы представления экземпляров
4. Узнаем о возможностях переопределения математических операций
5. Разберёмся со сравнением экземпляров
6. Узнаем об обработке атрибутов

Дополнительные материалы к лекции

Руководство по работе с дескриптором

<https://docs.python.org/3/howto/descriptor.html>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с созданием и удалением классов
2. Узнали о документировании классов
3. Изучили способы представления экземпляров
4. Узнали о возможностях переопределения математических операций
5. Разобрались со сравнением экземпляров
6. Узнали об обработке атрибутов

Термины лекции

- **Дескриптор** — это атрибут объекта со “связанным поведением”, то есть такой атрибут, при доступе к которому его поведение переопределяется методом протокола дескриптора. Эти методы `__get__`, `__set__` и `__delete__`. Если хотя бы один из этих методов определен в объекте, то можно сказать что этот метод дескриптор.

Подробный текст лекции

1. Класс как функция

При желании можно заставить класс, а точнее его экземпляры вести себя как функции. После имени экземпляра указываются круглые скобки с параметрами для вызова и экземпляр возвращает ответ. Разберём как это работает.

```
class Number:
    def __init__(self, num):
        self.num = num

n = Number(42)
print(f'{callable(Number) = }')
print(f'{callable(n) = }')
```

Класс Number имеет метод инициализации для сохранения числа. Мы создали экземпляр класса n и воспользовались встроенной функцией callable. Для класса получили истину, для экземпляра ложь. Функция отвечает на вопрос вызываемый перед нами объект или нет. Вызов класса возможен. Он запускает инициализацию и возвращает экземпляр. Вызвать экземпляр нельзя.

Метод вызова функции __call__

Создадим класс, экземпляры которого можно вызывать. Например для добавления очередного элемента во внутренний словарь класса по типам.

```
from collections import defaultdict

class Storage:
    def __init__(self):
        self.storage = defaultdict(list)

    def __str__(self):
        txt = '\n'.join((f'{k}: {v}' for k, v in
self.storage.items()))
        return f'Объекты хранилища по типам:\n{txt}'
```

```

def __call__(self, value):
    self.storage[type(value)].append(value)
    return f'К типу {type(value)} добавлен {value}'

s = Storage()
print(s(42))
print(s(72))
print(s('Hello world!'))
print(s(0))
print(s)

```

При создании класса используется продвинутая версия словаря из модуля collections — defaultdict. Словарю передана функция list. При обращении к несуществующему ключу вместо ошибки будет создан ключ и вызвана функция list для создания значения ключа.

Каждый вызов экземпляра добавляет переданный аргумент value в словарь storage и возвращает строку с информацией о выполненном действии.

Последовательно вызывая экземпляр с числами и текстом выводим его на печать и видим содержимое на момент печати.

Плюсом вызова экземпляра является то, что он не удаляется из памяти после вызова как обычная функция. Следовательно экземпляр может накапливать значения, использоваться в технологии мемоизации. Её рассматривали на лекции о декораторах.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```

class MyClass:

    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __repr__(self):
        return f'MyClass(a={self.a}, b={self.b})'

    def __call__(self, *args, **kwargs):

```

```
self.a.append(args)
self.b.update(kwargs)
return True
```

```
x = MyClass([42], {73: True})
y = x(3.14, 100, 500, start=1)
x(y=y)
print(x)
```

2. Создаём итераторы

Список list можно передать в цикл for in для перебора его элементов, итерации. Также итерироваться по списку можно в генераторных выражениях. А можно передать список функции для итерации, например функции all(). У итерируемых объектов много способов использования. Можно ли создать итерируемый объект самому? Да. Если экземпляр класса должен итерироваться, необходимо реализовать пару дандер методов.

Создадим класс экземпляра которого будет выдавать [числа Фибоначчи](#) в диапазоне начиная с числа больше или равного start и заканчивая числом меньше stop.

```
class Fibonacci:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
        self.first = 0
        self.second = 1

fib = Fibonacci(20, 100)
for num in fib: # TypeError: 'Fibonacci' object is not iterable
    print(num)
```

Внутри дандер __init__ запомнили границы start и stop и определили нулевое и первое число Фибоначчи в свойствах first и second соответственно.

Создание экземпляра не вызывает проблем. А попытка получить числа в цикле увенчалась ошибкой. Python сообщил, что объект не итерируемый.

Возврат итератора, `__iter__`

Для того, чтобы объект стал итерируемым, ему необходимо вернуть объект-итератор. В нашем случае экземпляр класса и есть объект-итератор. Следовательно он должен вернуть сам себя. Для возврата итератора нужно создать дандер метод `__iter__`.

```
class Fibonacci:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
        self.first = 0
        self.second = 1

    def __iter__(self):
        return self

fib = Fibonacci(20, 100)
for num in fib:    # TypeError: iter() returned non-iterator of
    print(num)    type 'Fibonacci'
```

Две строки метода вернули ссылку на самого себя. В результате получаем новую ошибку. Вернулся не итерируемый объект.

Возврат очередного значения, `__next__`

Как вы помните из лекции об итераторах и генераторах, любая итерация представляет из себя последовательный вызов функции `next()` с итератором в качестве аргумента.

Для возврата такого значения необходимо определить дандер метод `__next__`.

```
class Fibonacci:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
```

```

        self.first = 0
        self.second = 1

    def __iter__(self):
        return self

    def __next__(self):
        while self.first < self.stop:
            self.first, self.second = self.second, self.first +
self.second
            if self.start <= self.first < self.stop:
                return self.first
            raise StopIteration

fib = Fibonacci(20, 100)
for num in fib:
    print(num)

```

Итератор отработал как и ожидалось.

- дандер `__next__` создаёт цикл пока число в `first` не превысит значение `stop`
- получаем следующую пару Фибоначчи в `first` и `second`
- если `first` оказывается внутри диапазона `[start, stop)`, возвращаем очередной элемент
- обязательным условием для завершения итерации является вызов ошибки `StopIteration`. Python обрабатывает её как сигнал для завершения итерации и перехода к следующему за циклом коду. Остановки кода по ошибке не будет.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```

class Iter:
    def __init__(self, start, stop):
        self.start = start
        self.stop = stop

    def __iter__(self):
        return self

    def __next__(self):
        for i in range(self.start, self.stop):

```

```
        return chr(i)
    raise StopIteration
```

```
chars = Iter(65, 91)
for c in chars:
    print(c)
```

3. Создаём менеджер контекста with

Менеджер контекста with запускает два дандер метода. Один в момент вызова менеджера, а второй в момент выхода из внутреннего блока кода. Знакомая нам функция open() поддерживает работу с менеджером контекста. При вызове менеджера функция возвращает файловый дескриптор. А при выходе из него закрывает файл. Подобный функционал можно реализовать для любого объекта, где нужны одинаковые действия в начале и в конце. Рассмотрим пример работы с базой данных sqlite.

```
import sqlite3

connection = sqlite3.connect('sqlite.db')
cursor = connection.cursor()
cursor.execute("""create table if not exists users(name,
age);""")
cursor.execute("""insert into users values ('Гвидо', 66);""")
connection.commit()
connection.close()
```

Получение соединения с базой данных и получение курсора из соединения — обязательное начало для работы с базой.

Подтверждение изменений вызовом commit() и закрытие соединения с базой — обязательные действия в конце работы с базой.

Можно держать соединение открытым и подтверждать коммитить изменения после каждого действия с базой. А можно создать менеджер контекста.

- **Действия при входе в менеджер контекста, __enter__**

Создадим класс DB для упрощения работы с базой данных.


```

import sqlite3

class DB:
    def __init__(self, name):
        self.name = name
        self.connection = None
        self.cursor = None

    def __enter__(self):
        self.connection = sqlite3.connect(self.name)
        self.cursor = self.connection.cursor()
        return self.cursor

db = DB('sqlite.db')
with db as cur: # AttributeError: __exit__
    cur.execute("""create table if not exists users(name,
age);""")
    cur.execute("""insert into users values ('Гвидо', 66);""")

```

Экземпляр класса хранит имя базы, которое задаём один раз при получении экземпляра. Дополнительно запоминаем соединение и курсор. В момент создания экземпляра им присваиваем None.

Дандер `__enter__` определяет действия при входе в менеджер контекста. В нашем случае это установление соединения с базой данных и получение курсора. Сам курсор возвращаем в менеджер в переменную после `as`.

Если запустить код, получим ошибку доступа к атрибуту. Менеджер отказывается работать без указания действий для выхода.

- **Действия при выходе из менеджера контекста, `__exit__`**

Добавим дандер метод `__exit__` и пропишем в нём операции, обязательные при завершении работы с базой данных.

```

import sqlite3

class DB:
    def __init__(self, name):
        self.name = name
        self.connection = None
        self.cursor = None

    def __enter__(self):
        self.connection = sqlite3.connect(self.name)

```

```

        self.cursor = self.connection.cursor()
        return self.cursor

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.connection.commit()
        self.connection.close()
        self.cursor = self.connection = None

db = DB('sqlite.db')
with db as cur:
    cur.execute("""create table if not exists users(name,
age);""")
    cur.execute("""insert into users values ('Гвидо', 66);""")

```

Внутри `__exit__` подтверждаем изменения, закрываем соединения с базой и обнуляем свойства экземпляра. Параметры дандер `__exit__` содержат информацию о типе и значении ошибки и трассировку, если она возникла внутри менеджера. Если ошибок не было, все три параметра содержат `None`.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```

class MyCls:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    def __enter__(self):
        self.full_name = self.first_name + ' ' + self.last_name
        return self

    def __exit__(self, exc_type, exc_val, exc_tb):
        self.full_name = self.full_name.upper()

x = MyCls('Гвидо ван', 'Россум')
with x as y:
    print(y.full_name)
    print(x.full_name)
print(y.full_name)

```

```
print(x.full_name)
```

4. Декоратор @property

На прошлой лекции мы работали с классом треугольник и поместили его свойства защищёнными, добавив символ подчёркивания в начале имени. Но что если доступ к свойству нужен. Хотя бы на чтение. Для этого отлично подойдёт функция декоратор `property()`. Рассмотрим на более простом и коротком примере.

```
class User:
    def __init__(self, name):
        self._name = name

    @property
    def name(self):
        return self._name

user = User('Стивен')
print(f'{user.name = }')
user.name = 'Славик'  # AttributeError: can't set attribute 'name'
```

Класс `User` получает имя пользователя и сохраняет его в защищённой переменной экземпляра `_name`.

Далее создали метод `name` который возвращает значение из защищённого свойства `_name`. К методу применён декоратор `property`. Теперь Python воспринимает `name` не как имя вызываемого метода, а как название свойства.

При обращении к свойству `name` получаем результат — имя пользователя. Если же сделать попытку на изменение свойства, получим ошибку.

- **Getter**

Декоратор `property` позволяет создавать “геттеры”. Это методы, которые выдают себя за свойства, позволяют прочесть результат, но блокируют возможность записи. Рассмотрим другой пример “геттера”.

```
class User:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
```

```

        self.last_name = last_name

    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'

user = User('Стивен', 'Спилберг')
print(f'{user.first_name      =      }\n{user.last_name      = }
\n{user.full_name = }')
user.full_name = 'Стивен Хокинг'      # AttributeError: can't set
attribute 'full_name'
user.last_name = 'Хокинг'
print(f'{user.first_name      =      }\n{user.last_name      = }
\n{user.full_name = }')

```

Теперь у пользователя есть два публичных свойства для имени и фамилии. Кроме того есть свойство (а не метод) для вывода полного имени, т.е. с фамилией. Все три свойства работают на чтение. А вот перезаписать полное имя мы не можем. Зато ничего не мешает изменить фамилию и получить обновлённое полное имя.

- **Setter**

Python позволяет к “геттеру” добавить “сеттер” — метод контролирующий изменение свойства. Добавим пользователю возраст и будем контролировать чтобы новый возраст был больше старого. Например мы вручную обновляем данные раз в 5-10 лет.

```

class User:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
        self._age = 0

    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value > self._age:

```

```

        self._age = value
    else:
        raise ValueError(f'Новый возраст должен быть больше
текущего: {self._age}')

user = User('Стивен', 'Спилберг')
user.age = 75
print(f'Меня зовут {user.full_name} и мне {user.age} лет.')
print('Прошёл один год.')
user.age = 76
print(f'Меня зовут {user.full_name} и мне {user.age} лет.')
print('Прошло несколько лет. Изобретена технология омоложения. Но
возраст она не уменьшает.')
user.age = 25    # ValueError: Новый возраст должен быть больше
текущего: 76

```

Что получилось:

1. защищенное свойство `_age` получает значение ноль при рождении экземпляра, в дандер `__init__`
2. используя декоратор `property` создали свойство `age` для чтения текущего возраста
3. создаём “сеттер” для контроля записи новых значений в свойство `_age`
 - применяем декоратор `@age.setter`. Имя между `@` и точкой должно совпадать с именем “геттера”.
 - методу присваиваем такое же имя как и у свойства и он должен принимать значения помимо `self`
 - внутри метода делаем проверку на увеличения возраста
 - если возраст увеличивается, обновляем свойство `_age`
 - если возраст не увеличился вызываем ошибку `ValueError` и сообщаем её причину
4. В основном коде за просто увеличиваем возраст пользователя, но не можем его уменьшить.

При создании “сеттера” не обязательно вызывать ошибки. В целом внутри может быть прописана любая логика. Например вы работаете с финансовой программой и присваиваете новую сумму денег. “Сеттер” будет приводить сумму к типу `Decimal` перед присваиванием.

• Deleter

Помимо “геттера” и “сеттера” можно создать “делейтер”. Он выполняется при вызове команды `del` для свойства. Один из возможных вариантов использования “делейтера” - заменять значение на какое-то по умолчанию или помечать элемент

скрытым вместо удаления.

```
class User:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name
        self._age = 0

    @property
    def full_name(self):
        return f'{self.first_name} {self.last_name}'

    @property
    def age(self):
        return self._age

    @age.setter
    def age(self, value):
        if value > self._age:
            self._age = value
        else:
            raise ValueError(f'Новый возраст должен быть больше
текущего: {self._age}')

    @age.deleter
    def age(self):
        self._age = 0

user = User('Стивен', 'Спилберг')
user.age = 75
print(f'Меня зовут {user.full_name} и мне {user.age} лет.')
print('Прошло много лет. Изобретена технология перерождения.')
del user.age
print(f'Меня зовут {user.full_name} и мне {user.age} лет.')
```

Создание “делейтера” аналогично “сеттеру”. Также используется декоратор с именем свойства, но после точки пишем deleter. Внутри метода прописываются действия для удаления.

Антипаттерн геттера, сеттера, делейтера

Представленный ниже код является кодом ради кода и не имеет смысла в языке Python. Избегайте подобного. И да, код работает верно. Просто он не делает ничего нового.

```
class BadPattern:
    def __init__(self, x):
        self._x = x

    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x
```

Все три декоратора ничего не делают. Подобный код в Python должен выглядеть так, без защиты переменной x

```
class GoodPattern:
    def __init__(self, x):
        self.x = x
```

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class MyCls:
    def __init__(self, first_name, last_name):
        self.first_name = first_name
        self.last_name = last_name

    @property
    def full_name(self):
        return self.first_name + ' ' + self.last_name

    @full_name.setter
```

```
def full_name(self, value: str):
    self.first_name, self.last_name, _ = value.split()

x = MyCls('Стивен', 'Хокинг')
print(x.full_name)
x.full_name = 'Гвидо ван Россум'
print(x.full_name)
```

5. Дескрипторы

Дескриптор - это атрибут объекта со “связанным поведением”, то есть такой атрибут, при доступе к которому его поведение переопределяется методом протокола дескриптора. Эти методы `__get__`, `__set__` и `__delete__`. Если хотя бы один из этих методов определен в объекте, то можно сказать что этот метод дескриптор.

Звучит немного сложно. Так и есть. Дескрипторы не нужны для простых классов. Их польза проявляется при метапрограммировании, создании фреймворков.

Посмотрите на то как в Django создаются модели для работы с базой данных. Пример взят из [официальной документации](#)

```
from django.db import models

class Person(models.Model):
    first_name = models.CharField(max_length=30)
    last_name = models.CharField(max_length=30)
```

Как и почему работает код, где на уровне класса в обход инициализации создаются два свойства как экземпляры другого класса? Под капотом работают дескрипторы. Напишем класс, который хранит имя ученика, его возраст, номер класса (от 1 до 11) и номер кабинета, в котором сидит класс.

```
class Student:
    def __init__(self, name, age, grade, office):
        self.name = name
        self.age = age
        self.grade = grade
        self.office = office
```



```

def __repr__(self):
    return f'Student(name={self.name}, age={self.age},
grade={self.grade}, office={self.office})'

std1 = Student('Шурик', 7, 1, 12)
print(std1)

```

А теперь внимательно посмотрим на числовые значения.

- возраст должен быть больше нуля
- класс должен быть от 1 до 11
- кабинет должен быть номером в каком-то диапазоне. Предположим, что в нашей школе кабинеты нумеруются от 3 до 42

Ничего не мешает добавить “сеттеры” для каждого из свойств. Декоратор `property` мы уже прошли. Но если присмотреться, в трёх случаях мы задаём диапазон для целого числа. Дескрипторы позволяют сделать проверку на диапазон один раз и использовать её для всех трёх свойств. Ниже полностью готовый код.

```

class Range:
    def __init__(self, min_value: int = None, max_value: int = None):
        self.min_value = min_value
        self.max_value = max_value

    def __set_name__(self, owner, name):
        self.param_name = '_' + name

    def __get__(self, instance, owner):
        return getattr(instance, self.param_name)

    def __set__(self, instance, value):
        self.validate(value)
        setattr(instance, self.param_name, value)

    def __delete__(self, instance):
        raise AttributeError(f'Свойство "{self.param_name}" нельзя
удалить')

    def validate(self, value):
        if not isinstance(value, int):
            raise TypeError(f'Значение {value} должно быть целым
числом')
        if self.min_value is not None and value < self.min_value:
            raise ValueError(f'Значение {value} должно быть больше

```

```

или равно {self.min_value}')
    if self.max_value is not None and value >= self.max_value:
        raise ValueError(f'Значение {value} должно быть меньше
{self.max_value}')
```

```

class Student:
    age = Range(3, 103)
    grade = Range(1, 11 + 1)
    office = Range(3, 42 + 1)

    def __init__(self, name, age, grade, office):
        self.name = name
        self.age = age
        self.grade = grade
        self.office = office

    def __repr__(self):
        return f'Student(name={self.name}, age={self.age},
grade={self.grade}, office={self.office})'

if __name__ == '__main__':
    std_one = Student('Архимед', 12, 4, 29)
    std_other = Student('Аристотель', 2406, 5, 17) # ValueError:
Значение 2406 должно быть меньше 103
    print(f'{std_one} = { }')
    std_one.age = 15
    print(f'{std_one} = { }')
    std_one.grade = 11.0 # TypeError: Значение 11.0 должно быть
целым числом
    std_one.office = 73 # ValueError: Значение 73 должно быть
меньше 42
    del std_one.age # AttributeError: Свойство "_age" нельзя
удалять
    print(f'{std_one.__dict__} = { }')
```

Разберём код сверху вниз.

Класс-дескриптор Range

Мы создали дескриптор, реализующий все базовые дандер методы: чтения, записи и удаления.

В первую очередь инициализируем класс с параметрами для минимального и максимального значения. По умолчанию они равны None, любая из границ может быть открытой. Если же значения переданы, будем следовать правилу функции range - левая граница входит, а правая не входит.

- **Контроль имён, `__set_name__`**

Следующий метод срабатывает при определении имён свойств. В нашем случае это переменных уровня класса, определённые сразу после заголовка класса Student. Обратите внимание на локальную переменную `param_name`, которая получает имя создаваемой переменной с символом подчёркивания в начале. Дандер занимается инкапсуляцией за нас.

- **Контроль получения значений, `__get__`**

Всего одна строчка кода использует функцию `getattr()` для получения у объекта `instance` значения для свойства `self.param_name`, того самого с подчёркиванием в начале. Мы ничего не меняем, а лишь возвращаем значение свойства экземпляра.

- **Контроль присвоение значений, `__set__`**

Дандер ничего не возвращает. В первой строк вызываем метод `validate`. Он отвечает за попадание целого числа в диапазон, заданный при инициализации. Вторая строка сработает в том случае, когда валидация пройдена успешно и не вызвала ошибки. В этом случае функция `setattr()` присваивает у экземпляра `instance` параметру `self.param_name` значение `value`. Это значение стоит справа от знака равно в основном коде.

Рассмотрим подробнее работу метода `validate`:

- метод принимает значение `value` и выполняет ряд проверок с ним
- если значение не является целым числом, вызываем ошибку `TypeError`
- если задана нижняя граница и значение меньше неё, вызываем ошибку `ValueError`
- аналогично если задана верхняя граница и значение больше или равно границе, вызываем ошибку `ValueError`
- в случае прохождения всех проверок метод ничего не возвращает. Он позволяет выполняться коду дальше

- **Контроль удаления атрибутов, `__delete__`**

Как понятно из названия метод срабатывает при попытке удалить свойство командой `del`. В нашем примере вызываем ошибку `AttributeError` и ничего не удаляем.

Класс Student

Далее создаём класс для хранения информации о студентах. На уровне класса задаём три переменные, которые являются экземплярами класса Range. Для этих свойств будут срабатывать методы дескриптора.

При инициализации студента получаем имя и три уже описанных параметра. Отдельно задаём дандер `__repr__` для вывода на печать. Ничего нового тут нет.

В основном блоке кода с лёгкостью создаём первого студента. Все атрибуты прошли проверку. А вот создать второго студента не получилось, его возраст вышел за пределы диапазона. Как видите дескриптор работает уже на этапе инициализации экземпляра.

Если сделать изменение в пределах допустимого, код работает как обычно. А при попытке присвоения значения не проходящего валидацию получаем соответствующую ошибку.

Присмотритесь к ошибке при попытке удалить свойство `age`. Дескриптор сообщает, что `_age` нельзя удалить. Т.е. свойства скрыты от нас, но мы можем обращаться используя обычные имена, без подчёркивания в начале.

В финальной строке смотрим дандер переменную `__dict__` и видим, что наши свойства с диапазонами начинаются с подчёркивания. Результат работы дандер `__set_name__`.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
class Text:
    def __init__(self, param):
        self.param = param

    def __set_name__(self, owner, name):
        self.param_name = '_' + name

    def __set__(self, instance, value):
        if self.param(value):
            setattr(instance, self.param_name, value)
        else:
            raise ValueError(f'Bad {value}')

class User:
```

```

first_name = Text(str.istitle)
last_name = Text(str.isupper)

def __init__(self, first_name, last_name):
    self.first_name = first_name
    self.last_name = last_name

def __repr__(self):
    return f'Student(age={self.first_name},
grade={self.last_name})'

if __name__ == '__main__':
    std_one = User('Гвидо ван', 'Россум')

```

6. Экономим память

Мы уже несколько раз сталкивались с дандер словарём `__dict__`. Его предназначение — хранить атрибуты и их значения у каждого объекта Python.

Хранитель атрибутов `__dict__`

Рассмотрим уже знакомый по прошлой лекции класс `Triangle` и выведем на печать содержимое `__dict__` у экземпляра и у класса.

```

from math import sqrt

class Triangle:
    def __init__(self, a, b, c):
        self._a = a
        self._b = b
        self._c = c

    def __str__(self):
        return f'Треугольник со сторонами: {self._a}, {self._b}, {self._c}'

    def __repr__(self):
        return f'Triangle({self._a}, {self._b}, {self._c})'

```

```

def __eq__(self, other):
    first = sorted((self._a, self._b, self._c))
    second = sorted((other._a, other._b, other._c))
    return first == second

def area(self):
    p = (self._a + self._b + self._c) / 2
    _area = sqrt(p * (p - self._a) * (p - self._b) * (p -
self._c))
    return _area

def __lt__(self, other):
    return self.area() < other.area()

def __hash__(self):
    return hash((self._a, self._b, self._c))

triangle = Triangle(3, 4, 5)
print(triangle)
print(triangle.__dict__)
print(Triangle.__dict__)

```

Как видите экземпляр хранить лишь три свойства, определённые внутри метода инициализации. За всем остальным он обращается к своему классу.

Что касается класса, его словарь имени и адреса дандеров, методов и даже пустой дандер `__doc__`, ведь мы не сделали строку документации.

При редкой необходимости можно обращаться к ключам словаря для получения или изменения значений.

Экономия памяти, `__slots__`

При создании класса можно явно указать перечень имён свойств, которые в нём будут использоваться.

```

class Triangle:
    __slots__ = ('_a', '_b', '_c')

    def __init__(self, a, b, c):
        ...

```

Подобная запись говорит о том, что теперь у нас лишь три свойства. Python не позволит добавить новые.

А при попытке обратиться к словарю экземпляра получим ошибку `AttributeError`: `'Triangle' object has no attribute '__dict__'. Did you mean: '__dir__'?`

Коротко о том, что даёт замена изменяемого `__dict__` на неизменяемый `__slots__`?

1. Обеспечивает немедленное обнаружение ошибок из-за неправильного написания атрибутов. Допускаются только имена атрибутов, указанные в `__slots__`
2. Помогает создавать неизменяемые объекты, в которых дескрипторы управляют доступом к закрытым атрибутам, хранящимся в `__slots__`
3. Экономит память. В 64-битной сборке Linux экземпляр с двумя атрибутами занимает 48 байт со `__slots__` и 152 байт без него. Экономия памяти имеет значение только тогда, когда будет создано большое количество экземпляров.
4. Улучшает скорость. По данным на Python 3.10 на процессоре Apple M1 чтение переменных экземпляра выполняется на 35% быстрее со `__slots__`.
5. Блокирует такие инструменты как `functools.cached_property()`, которым для правильной работы требуется экземплярный словарь.

Вывод

На этой лекции мы:

1. Разобрались в превращении объекта в функцию
2. Изучили способы создания итераторов
3. Узнали о создании менеджеров контекста
4. Разобрались в превращении методов в свойства
5. Изучили работу дескрипторов
6. Узнали о способах экономии памяти

Краткий анонс следующей лекции

1. Разберёмся с обработкой ошибок в Python
2. Изучим иерархию встроенных исключений
3. Узнаем о способе принудительного поднятия исключения в коде
4. Разберёмся в создании собственных исключений

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий и попробуйте перенести переменные и функции в класс, если это не задачи про классы. Добавьте к ним дандер методы из лекции для решения исходной задачи.

Оглавление

На этой лекции мы	2
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	3
Подробный текст лекции	
1. Обработка исключительных ситуаций в Python	3
Команда try	5
Цикл while для обработки ошибок ввода	6
Несколько except для одного try	7
Команда else	8
Вложенные блоки обработки исключений	9
Блок finally без except	10
BaseException и его ближайшие родственники	13
4. Создание собственных исключений	18
Вывод	21

На этой лекции мы

1. Разберёмся с обработкой ошибок в Python
2. Изучим иерархию встроенных исключений
3. Узнаем о способе принудительного поднятия исключения в коде
4. Разберёмся в создании собственных исключений

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались в превращении объекта в функцию

2. Изучили способы создания итераторов
3. Узнали о создании менеджеров контекста
4. Разобрались в превращении методов в свойства
5. Изучили работу дескрипторов
6. Узнали о способах экономии памяти

Термины лекции

- **Исключение** — это любое состояние ошибки или непредвиденное поведение, возникающее при выполнении программы

Подробный текст лекции

1. Обработка исключительных ситуаций в Python

На протяжении курса мы регулярно сталкивались с ошибками в программах. Python выдавал несколько строк красного текста в консоль — результат трассировки ошибки. И после такого программа переставала работать.

Встроенный в язык механизм обработки исключений позволяет изменить поведение программы при появлении ошибки. Впрочем, некоторые ошибки действительно должны завершать программу. О них подробнее чуть позже в этой лекции.

А пока рассмотрим простой пример кода и результат его выполнения

Код:

```
num = int(input('Введите целое число: '))
```

Результат

```
Введите целое число: сорок два
Traceback (most recent call last):
  File "C:\Users\main.py", line 1, in <module>
    num = int(input('Введите целое число: '))
ValueError: invalid literal for int() with base 10: 'сорок два'
```

```
Process finished with exit code 1
```

Программа запросила целое число, пользователь написал “сорок два” и мы получили ошибку `ValueError`. Красный текст — результат трассировки ошибки. Читать его лучше построчно снизу вверх. Самая нижняя строка указывает какую именно ошибку мы получили и почему. Строкой выше указывается строка кода, повлёкшая ошибку. Отдельно Python указывает название файла с номером строки для быстрого перехода к месту ошибки. При этом строчек с кодом и указаниями на строки ошибки может быть несколько. Зависит от того как долго ошибка распространялась по цепочке кода.

Посмотрите на код ниже.

```
def get(text: str = None) -> int:
    data = input(text)
    num = int(data)
    return num

if __name__ == '__main__':
    number = get('Введите целый делитель: ')
    print(f'100 / {number} = {100 / number}')
```

Попробуем и тут ввести “сорок два”

```
Введите целый делитель: сорок два
Traceback (most recent call last):
  File "C:\Users\main.py", line 8, in <module>
    number = get('Введите целый делитель: ')
  File "C:\Users\main.py", line 3, in get
    num = int(data)
ValueError: invalid literal for int() with base 10: 'сорок два'

Process finished with exit code 1
```

Ошибка точно такая же. Но теперь трассировка показывает, что ошибка возникла в третьей строке кода при приведении данных к целому типу. А чуть ранее мы вызвали функцию `get` в строке 8, которая и заставила выполняться третью строчку кода.

Чтение трассировки помогает найти источник ошибки. Python и тут проявляет максимальное дружелюбие к разработчику и даёт максимально возможное количество информации об ошибке. Если вы писали на других языках, скорее всего вы уже заметили это дружелюбие.

Команда try

Что делать, если мы не хотим завершать программу в случае появления ошибки? Команда try позволяет обернуть блок кода с возможной ошибкой и отловить её.

Поступим так с третьей строкой нашей программы.

```
def get(text: str = None) -> int:
    data = input(text)
    try:
        num = int(data)
    return num

if __name__ == '__main__':
    number = get('Введите целый делитель: ')
    print(f'100 / {number} = {100 / number}')
```

Команда try: должна завершать строку двоеточием, а вложенный блок кода пишется с отступами. Всё точно так же как с if, for, while и т.п.



Важно! Блок кода внутри try в идеале должен состоять из одной строки кода — потенциального источника ошибки. Оборачивание всей программы в блок try считается плохим стилем программирования.

Запустим код и получим самую частую ошибку новичка — SyntaxError.

```
File "C:\Users\main.py", line 5
    return num
    ^^^^^
SyntaxError: expected 'except' or 'finally' block
```

Ошибка синтаксиса говорит о том, что Python не понял вас. “Дружище, твой код не похож на Python программу. Я указал на место, откуда надо начинать искать ошибку. Но она может быть и раньше этой строки”. Синтаксические ошибки обязательно исправлять в процессе создания программы, а не пытаться их обрабатывать.

Команда except

Как вы верно догадались из текста синтаксической ошибки, команда try должна работать в связке с командой except или finally.

Рассмотрим вариант обработки ошибки в нашем коде:

```
def get(text: str = None) -> int:
    data = input(text)
    try:
        num = int(data)
    except ValueError as e:
        print(f'Ваш ввод привёл к ошибке ValueError: {e}')
        num = 1
        print(f'Будем считать результатом ввода число {num}')
    return num

if __name__ == '__main__':
    number = get('Введите целый делитель: ')
    print(f'100 / {number} = {100 / number}')
```

После зарезервированного слова except указывается класс ошибки, которую мы хотим обработать. Обычно используется запись вида except NameError as e: Таким образом в переменную e попадает информация об ошибке. Например для вывода её в консоль, сохранения в логи и т.п. В нашем случае при невозможности получить целое число из пользовательского ввода сообщаем ему об этом. Далее делаем вид, что была введена единица. Программа продолжает работать несмотря на ошибку.

- **Цикл while для обработки ошибок ввода**

Иногда необходимо получить данные повторно, если попытка не удалась. В случае с пользователем можем спрашивать его бесконечно, пока не добъёмся ввода целого числа.

```
def get(text: str = None) -> int:
    while True:
        try:
            num = int(input(text))
            break
        except ValueError as e:
            print(f'Ваш ввод привёл к ошибке ValueError: {e}\n'
                  f'Попробуйте снова')
    return num

if __name__ == '__main__':
```

```
number = get('Введите целый делитель: ')
print(f'100 / {number} = {100 / number}')
```

Бесконечный цикл `while True` можно прервать командой `break`. В случае преобразования ввода пользователя к целому без ошибок она завершит цикл и вернёт число из функции. При появлении ошибки дальнейшие строки не выполняются, сразу переходим в блок `except`. Обработав ошибку мы возвращаемся к началу цикла, следовательно повторяем запрос.

Подобное поведение можно применить не только к функции `input`, но и к любой ситуации получения данных. Код ниже имитирует получение данных из источника (база данных, сайт, удалённые сервер и т.п.).

```
MAX_COUNT = 5

count = 0
while count < MAX_COUNT:
    count += 1
    try:
        data = get_data()
        break
    except ConnectionError as e:
        print(f'Попытка {count} из {MAX_COUNT} завершилась ошибкой {e}')
```

- **Несколько `except` для одного `try`**

Как вы уже догадались при вводе нуля в примерах на деление выше мы получим ошибку. Это `ZeroDivisionError: division by zero`.

Вспомним высшую математику, а именно то, что при делении любого числа на ноль получаем бесконечность.

```
def hundred_div_num(text: str = None) -> tuple[int, float]:
    while True:
        try:
            num = int(input(text))
            div = 100 / num
            break
        except ValueError as e:
            print(f'Ваш ввод привёл к ошибке ValueError: {e}\n'
                  f'Попробуйте снова')
        except ZeroDivisionError as e:
            div = float('inf')
            break
    return num, div
```

```
if __name__ == '__main__':
    n, d = hundred_div_num('Введите целый делитель: ')
    print(f'Результат операции: "100 / {n} = {d}"')
```

В приведённом примере блок try обрабатывает сразу несколько строчек, которые способны вызвать разные ошибки. Не лучший вариант. Правильнее было бы разделить код на отдельные try блоки со своими возможными исключениями. Но зато мы познакомились с обработкой нескольких ошибок разом.

Если не удалось получить целое число, обрабатываем ошибку значения и даём ещё один шанс. А если делим на ноль, вместо ошибки возвращаем бесконечность.

Внимание! Язык Python поддерживает такие математические числа как бесконечность и минус бесконечность. Записываются они как особая форма вещественного числа:

- float('inf') - бесконечность
- float('-inf') - минус бесконечность

Команда else

Если надо выполнить код только в случае успешного завершения блока try, можно воспользоваться командой else в связке try-except-else

```
MAX_COUNT = 5

result = None
for count in range(1, MAX_COUNT + 1):
    try:
        num = int(input('Введите целое число: '))
        print('Успешно получили целое число')
    except ValueError as e:
        print(f'Попытка {count} из {MAX_COUNT} завершилась ошибкой {e}')
    else:
        result = 100 / num
        break

print(f'{result = }')
```

В приведённом примере пользователь вводит число. Если получаем ошибку, сообщаем о ней пользователю. Всего даём MAX_COUNT попыток. Но стоит успешно преобразовать текст в целое как сработает блок else. Он сохранит результат деления и завершит цикл попыток.

Если внутри блока try произойдёт одно из событий ниже, блок else не будет вызван:

- возбуждено исключение
- выполнена команда return
- выполнена команда break
- выполнена команда continue

Именно по этой причине в нашем примере break переключал из try в else.

Блок else может быть лишь один и обязан следовать за блоком except.

Вложенные блоки обработки исключений

При необходимости одни try блоки могут включать другие. Аналогично работают вложенные циклы или вложенные if — сложные ветвления.

Перепишем код выше так, чтобы ошибка деления на ноль обрабатывалась внутри блока else верхнего try.

```
MAX_COUNT = 5

result = None
for count in range(1, MAX_COUNT + 1):
    try:
        num = int(input('Введите целое число: '))
        print('Успешно получили целое число')
    except ValueError as e:
        print(f'Попытка {count} из {MAX_COUNT} завершилась ошибкой {e}')
    else:
        try:
            result = 100 / num
        except ZeroDivisionError as e:
            result = float('inf')
        break

print(f'{result = }')
```

Если удалось получить целое число, заглядываем в else и пытаемся делить. Но если деление на ноль, возвращаем бесконечность вместо ошибки.

Команда finally

Ещё одна команда для обработки исключений — finally. Она срабатывает во всех случаях. И если была ошибка и отработал блок except. И если ошибки не было.

```
def get(text: str = None) -> int:
    num = None
    try:
        num = int(input(text))
    except ValueError as e:
        print(f'Ваш ввод привёл к ошибке ValueError: {e}')
    finally:
        return num if isinstance(num, int) else 1

if __name__ == '__main__':
    number = get('Введите целый делитель: ')
    try:
        print(f'100 / {number} = {100 / number}')
    except ZeroDivisionError as e:
        print(f'На ноль делить нельзя. Получим {e}')
```

Даём пользователю одну попытку на ввод числа. Независимо от результата сработает блок finally. Он вернёт либо целое число, либо единицу, если получить целое не удалось. Обработку деления на ноль вынесли в основной код.

Обычно finally используют для действий, которые обязательны независимо от того была ошибка или нет.

Блок finally без except

Вполне допустимо использовать только связку try-finally. Например мы хотим прочитать информацию из файла. И независимо от результат чтения закрыть его.

```
file = open('text.txt', 'r', encoding='utf-8')
try:
    data = file.read().split()
    print(data[len(data)])
finally:
    print('Закрываю файл')
    file.close()
```

Открываем файл для чтения, скачиваем его и сохраняем каждую строку в отдельную ячейку списка. Но внезапно “забываем”, что нумерация начинается с нуля, а для доступа к последнему элементу можно использовать индекс -1. В результате попытка прочитать информацию из ячейки, следующей за последней

выдаёт ошибку `IndexError: list index out of range`. Но блок `finally` закрывает файл раньше, чем программа завершит свою работу.



Важно! Для ситуации выше правильным будет использовать менеджер контекста для гарантированного закрытия файла.

Как вы можете видеть комбинации `try-except-else-finally` дают большой простор для отлова исключений и изменения логика кода в зависимости от ситуаций.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты. Четыре попытки ввода пользователя указаны ниже кода

```
d = {'42': 73}
try:
    key, value = input('Ключ и значение: ').split()
    if d[key] == value:
        r = 'Совпадение'
except ValueError as e:
    print(e)
except KeyError as e:
    print(e)
else:
    print(r)
finally:
    print(d)
```

```
>>> Ключ и значение: 42 13
>>> Ключ и значение: 42 73 3
>>> Ключ и значение: 73 42
>>> Ключ и значение: 42 73
```

2. Иерархия исключений в Python

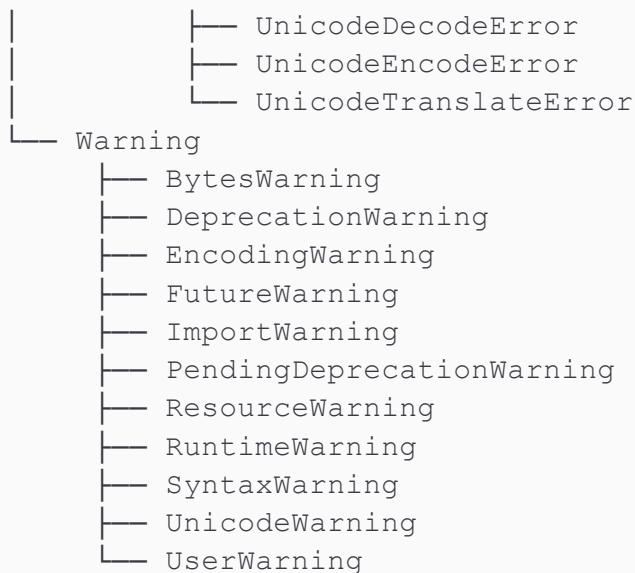
Любое исключение в Python является классом. При этом одни классы наследуются от других. Общая [иерархия классов для встроенных исключений Python](#) выглядит так:

```
BaseException
```

```

├─ BaseExceptionGroup
├─ GeneratorExit
├─ KeyboardInterrupt
├─ SystemExit
└─ Exception
    ├── ArithmeticError
    │   ├── FloatingPointError
    │   ├── OverflowError
    │   └─ ZeroDivisionError
    ├── AssertionError
    ├── AttributeError
    ├── BufferError
    ├── EOFError
    ├── ExceptionGroup [BaseExceptionGroup]
    ├── ImportError
    │   └─ ModuleNotFoundError
    ├── LookupError
    │   ├── IndexError
    │   └─ KeyError
    ├── MemoryError
    ├── NameError
    │   └─ UnboundLocalError
    ├── OSError
    │   ├── BlockingIOError
    │   ├── ChildProcessError
    │   ├── ConnectionError
    │   │   ├── BrokenPipeError
    │   │   ├── ConnectionAbortedError
    │   │   ├── ConnectionRefusedError
    │   │   └─ ConnectionResetError
    │   ├── FileExistsError
    │   ├── FileNotFoundError
    │   ├── InterruptedError
    │   ├── IsADirectoryError
    │   ├── NotADirectoryError
    │   ├── PermissionError
    │   ├── ProcessLookupError
    │   └─ TimeoutError
    ├── ReferenceError
    ├── RuntimeError
    │   ├── NotImplementedError
    │   └─ RecursionError
    ├── StopAsyncIteration
    ├── StopIteration
    ├── SyntaxError
    │   ├── IndentationError
    │   └─ TabError
    ├── SystemError
    ├── TypeError
    ├── ValueError
    │   └─ UnicodeError

```



BaseException и его ближайшие родственники

Базовым исключением является класс `BaseException`. Все остальные исключения являются его потомками. Не рекомендуется его перехватывать или использовать в создании своих исключений. Оно нужно для правильной иерархии исключений, ожидаемого поведения Python.

Помимо `BaseException` в подавляющем большинстве случаев не перехватывают и следующие исключения:

- **`BaseExceptionGroup`** — базовое исключение для объединения исключений в группу
- **`GeneratorExit`** — исключение создаёт генератор или корутину при закрытии. Подобное ситуация не является ошибкой с технической точки зрения.
- **`KeyboardInterrupt`** — возникает при нажатии комбинации клавиш, прерывающих работу программы. Например `Ctrl-C` в терминале посылает сигнал, требующий завершить процесс
- **`SystemExit`** — при выходе из программы через функцию `exit()` поднимается данное исключение.

Основной Exception и его наследники

Исключение Exception является базовым для всех исключений Python. Также оно используется при создании своих собственных исключений.

Перечислим некоторые часто встречающиеся исключения и дадим пояснения по ним.

- **ArithmeticError** — арифметическая ошибка.
 - **FloatingPointError** — порождается при неудачном выполнении операции с плавающей запятой. На практике встречается нечасто.
 - **OverflowError** — возникает, когда результат арифметической операции слишком велик для представления. Не появляется при обычной работе с целыми числами (так как python поддерживает длинные числа), но может возникать в некоторых других случаях.
 - **ZeroDivisionError** — деление на ноль.
- **AssertionError** — выражение в функции assert ложно. Подробнее об assert поговорим на следующей лекции, когда будем разбирать тестирование кода.
- **AttributeError** — объект не имеет данного атрибута (значения или метода).
- **BufferError** — операция, связанная с буфером, не может быть выполнена.
- **EOFError** — функция наткнулась на конец файла и не смогла прочитать то, что хотела.
- **ImportError** — не удалось импортирование модуля или его атрибута.
- **LookupError** — некорректный индекс или ключ.
 - **IndexError** — индекс не входит в диапазон элементов.
 - **KeyError** — несуществующий ключ (в словаре, множестве или другом объекте).
- **MemoryError** — недостаточно памяти.
- **NameError** — не найдено переменной с таким именем.
- **UnboundLocalError** — сделана ссылка на локальную переменную в функции, но переменная не определена ранее.
- **OSError** — ошибка, связанная с системой.
 - **BlockingIOError** — возникает, когда операция блокирует объект (например, сокет), установленный для неблокирующей операции
 - **ChildProcessError** — неудача при операции с дочерним процессом.
 - **ConnectionError** — базовый класс для исключений, связанных с подключениями.
 - **BrokenPipeError** — возникает при попытке записи в канал, в то время как другой конец был закрыт, или при попытке записи в сокет, который был отключен для записи
 - **ConnectionAbortedError** — попытка соединения прерывается узлом
 - **ConnectionRefusedError** — партнер отклоняет попытку подключения
 - **ConnectionResetError** - соединение сбрасывается узлом

- `FileExistsError` — попытка создания файла или директории, которая уже существует.
- `FileNotFoundError` — файл или директория не существует.
- `InterruptedError` — системный вызов прерван входящим сигналом.
- `IsADirectoryError` — ожидался файл, но это директория.
- `NotADirectoryError` — ожидалась директория, но это файл.
- `PermissionError` — не хватает прав доступа.
- `ProcessLookupError` — указанного процесса не существует.
- `TimeoutError` — закончилось время ожидания.
- `ReferenceError` — попытка доступа к атрибуту со слабой ссылкой.
- `RuntimeError` — возникает, когда исключение не попадает ни под одну из других категорий.
 - `NotImplementedError` — возникает, когда абстрактные методы класса требуют переопределения в дочерних классах.
 - `RecursionError` — превышена глубина стека вызова функций.
- `StopAsyncIteration` — порождается в корутине (асинхронной функции), если в итераторе больше нет элементов.
- `StopIteration` — порождается встроенной функцией `next`, если в итераторе больше нет элементов.
- `SyntaxError` — синтаксическая ошибка.
 - `IndentationError` — неправильные отступы.
 - `TabError` — смешивание в отступах табуляции и пробелов.
- `SystemError` — интерпретатор находит внутреннюю ошибку, но ситуация не выглядит настолько серьезной, чтобы заставить его отказаться от всякой надежды. Возвращаемое значение представляет собой строку, указывающую, что пошло не так.
- `TypeError` — операция применена к объекту несоответствующего типа.
- `ValueError` — функция получает аргумент правильного типа, но некорректного значения.
 - `UnicodeError` — ошибка, связанная с кодированием / раскодированием `unicode` в строках.
 - `UnicodeEncodeError` — исключение, связанное с кодированием `unicode`.
 - `UnicodeDecodeError` — исключение, связанное с декодированием `unicode`.
 - `UnicodeTranslateError` — исключение, связанное с переводом `unicode`.
- `Warning` — группа для предупреждений.

Группа Warning

Warning включает в себя ряд базовых предупреждений. Предупреждающие сообщения обычно выдаются в ситуациях, когда полезно предупредить пользователя о каком-либо состоянии в программе, когда это условие не требует возбуждения исключения и завершения программы. Например, может потребоваться выдать предупреждение, когда программа использует устаревший модуль.

3. Ключевое слово raise

Отдельно рассмотрим команду raise для поднятия исключений. С ней мы уже сталкивались на прошлой лекции. Команда поднимает исключение, указанное после неё. Используется для случаев, когда вы явно хотите сообщить о неправильной работе вашего кода.

Например у нас есть функция, которая запрещает изменять значение у существующих ключей.

```
def add_key(dct, key, value):
    if key in dct:
        raise KeyError(f'Перезаписывание существующего ключа
запрещено. {dct[key] = }')
    dct[key] = value
    return dct

data = {'one': 1, 'two': 2}
print(add_key(data, 'three', 3))
print(add_key(data, 'three', 3))
```

Первый раз мы смогли добавить пару ключ-значение в словарь. Но строкой ниже получили ошибку. Ключ уже добавлен в словарь и изменить его нельзя. Поднимаем исключение KeyError.

Поднимаем исключения в классах

Более сложный пример поднятия исключений — контроль создания класса. Подобное было при создании дескрипторов. В нашем случае класс отслеживает переданные аргументы в методе инициализации.

```
class User:
    def __init__(self, name, age):
        if 6 < len(name) < 30:
            self.name = name
        else:
            raise ValueError(f'Длина имени должна быть между 6 и
30 символами. {len(name) = }')
        if not isinstance(age, (int, float)):
            raise TypeError(f'Возраст должен быть числом.
Используйте int или float. {type(age) = }')
        elif age < 0:
            raise ValueError(f'Возраст должен быть положительным.
{age = }')
        else:
            self.age = age

user = User('Яков', '-12')
```

Класс контролирует длину имени пользователя. Далее убеждаемся, что возраст — число. И если верно, то проверяем больше ли нуля число.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
def func(a, b, c):
    if a < b < c:
        raise ValueError('Не перемешано!')
    elif sum((a, b, c)) == 42:
        raise NameError('Это имя занято!')
    elif max(a, b, c, key=len) < 5:
        raise MemoryError('Слишком глуп!')
    else:
        raise RuntimeError('Что-то не так!!!')

func(11, 7, 3) # 1
```



```
func(3, 2, 3)    # 2
func(73, -40, 9) # 3
func(10, 20, 30) # 4
```

4. Создание собственных исключений

В финале пару примеров создания собственных исключений. Попробуем для класса User из прошлого примера создать свои исключения.

Писать код исключений будем в отдельном файле error.py Начнём с того, что создадим своё собственное базовое исключение. От него будут наследоваться остальные наши исключения. Родительское исключение займёт пару строк кода

```
class UserException(Exception):
    pass
```

А теперь добавим исключения для ошибок имени и возраста пользователя. Пока это будут исключения на минималках.

```
class UserAgeError(UserException):
    pass

class UserNameError(UserException):
    pass
```

Теперь внесём правки в код инициализации пользователя. Заодно избавимся от магических чисел для минимальной и максимальной длины имени.

```
from error import UserNameError, UserAgeError

class User:
    MIN_LEN = 6
    MAX_LEN = 30

    def __init__(self, name, age):
        if self.MIN_LEN < len(name) < self.MAX_LEN:
            self.name = name
        else:
            raise UserNameError
        if not isinstance(age, (int, float)) or age < 0:
```

```
        raise UserAgeError
    else:
        self.age = age

user = User('Яков', '-12')
```

Подобный код отлично справляется с поставленной задачей. Но стал менее информативен в случае возникновения ошибок.

```
error.UserNameError
```

Понятно, что ошибка в имени. Но не очень информативно. Исправим ситуацию.

Методы `__init__` и `__str__` в классах своих исключений

Чтобы исключение давало подробную информацию об ошибке, будем передавать ему проблемную переменную. Класс `User` доработаем в строках подъёма ошибок.

```
from error import UserNameError, UserAgeError

class User:
    def __init__(self, name, age):
        if 6 < len(name) < 30:
            self.name = name
        else:
            raise UserNameError(name)
        if not isinstance(age, (int, float)) or age < 0:
            raise UserAgeError(age)
        else:
            self.age = age

user = User('Яков', '-12')
```

Благодаря наследованию переданные в исключение переменные могут выводиться в тексте ошибки.

```
raise UserNameError(name)
```

```
error.UserNameError: Яков
```

Уже лучше. Но без пары дандер методов в классах ошибок пока не идеально. Дорабатываем код в файле error.

```
class UserException(Exception):
    pass

class UserAgeError(UserException):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return f'Возраст пользователя должен быть целым int() или вещественным float() больше нуля.\n' \
               f'У вас тип {type(self.value)}, а значение {self.value}'

class UserNameError(UserException):
    def __init__(self, name, lower, upper):
        self.name = name
        self.lower = lower
        self.upper = upper

    def __str__(self):
        text = 'попадает в'
        if len(self.name) < self.lower:
            text = 'меньше нижней'
        elif len(self.name) > self.lower:
            text = 'больше верхней'
        return f'Имя пользователя {self.name} содержит {len(self.name)} символа(ов).\n' \
               f'Это {text} границы. Попадите в диапазон ({self.lower}, {self.upper}).'
```

В случае с возрастом просто получаем текущее значение в переменную value. Далее выводим информацию об ошибке без явного уточнения проблемы. Просто сообщаем о допустимых типе и значении, а также выводим переданное значение и его тип.

При обработке ошибки имени дополнительно принимаем в инициализацию граничные значения длины. Переменная text внутри дандер __str__ получает значение в зависимости от границы: “меньше нижней” или “больше верхней”. Вывод точно указывает на то, в какую из границ мы не попали.



Внимание! В классе User надо исправить строку вызова ошибки имени, чтобы код сработал верно. Иначе исключение вернёт нам исключение `TypeError: UsernameError.__init__() missing 2 required positional arguments: 'lower' and 'upper'`

```
...
def __init__(self, name, age):
    if self.MIN_LEN < len(name) < self.MAX_LEN:
        self.name = name
    else:
        raise UsernameError(name, self.MIN_LEN, self.MAX_LEN)
...
```

Вывод

На этой лекции мы:

1. Разобрались с обработкой ошибок в Python
2. Изучили иерархию встроенных исключений
3. Узнали о способе принудительного поднятия исключения в коде
4. Разобрались в создании собственных исключений

Краткий анонс следующей лекции

1. Разберёмся с написанием тестов в Python
2. Изучим возможности doctest
3. Узнаем о пакете для тестирования unittest
4. Разберёмся с тестированием через pytest

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий. Добавьте обработку исключений так, чтобы код продолжал работать и выполнял задачу. Для любителей сложностей создайте собственные исключения для каждой из задач.

Оглавление

На этой лекции мы	3
Дополнительные материалы к лекции	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Термины лекции	4
Подробный текст лекции	4
0. Основы тестирования	4
1. Основы doctest	5
Разработка через тестирование, TDD	8
Добавление теста	8
Запуск всех тестов: убедиться, что новые тесты не проходят	9
Написать код	10
Запуск всех тестов: убедиться, что все тесты проходят	11
Проверка исполняемой документации	11
Задание	13
2. Основы тестирования с unittest	14
Общие моменты работы с unittest	14
Сравнение тестов doctest и unittest	15
Кейс test_is_prime	17
Кейс test_type	17
Кейс test_value	17
Кейсы test_warning_false и test_warning_true	17
Запуск тестов doctest из unittest	17
Подготовка теста и сворачивание работ	18
Метод setUp	18
Метод tearDown	19
3. Основы тестирования с pytest	22
Команда assert	22
Общие моменты работы с pytest	23
Сравнение тестов pytest с doctest и unittest	24
Кейс test_is_prime	25
Кейс test_type	25
Кейс test_value	26
Кейс test_value_with_text	26
Кейсы test_warning_false и test_warning_true	26
Запуск тестов doctest и unittest	26

Фикстуры pytest как замены unittest setUp и tearDown	27
Ещё немного о фикстурах	28
Фикстура get_file	29
Фикстура set_num	29
Фикстура set_char	29
Кейс test_first_num	29
Кейс test_first_char	29
Задание	30
Вывод	30

На этой лекции мы

1. Разберёмся с написанием тестов в Python
2. Изучим возможности doctest
3. Узнаем о пакете для тестирования unittest
4. Разберёмся с тестированием через pytest

Дополнительные материалы к лекции

1. Библиотека mock объектов
<https://docs.python.org/3.11/library/unittest.mock.html>
2. Полная документация по pytest <https://docs.pytest.org/en/stable/contents.html>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с обработкой ошибок в Python
2. Изучили иерархию встроенных исключений
3. Узнали о способе принудительного поднятия исключения в коде
4. Разобрались в создании собственных исключений

Термины лекции

- **Тестирование программного обеспечения** — это исследование, проводимое с целью предоставления заинтересованным сторонам информации о качестве тестируемого программного продукта или услуги. Тестирование программного обеспечения также может обеспечить объективный, независимый взгляд на программное обеспечение, позволяющий бизнесу оценить и понять риски внедрения программного обеспечения.
- **Простое число** — натуральное число, имеющее ровно два различных натуральных делителя. Другими словами, натуральное число p является простым, если оно отлично от 1 и делится без остатка только на 1 и на само p .
- **Натуральные числа (от лат. *naturalis* «естественный»)** — числа, возникающие естественным образом при счёте (1, 2, 3, 4, 5, 6, 7 и так далее).
- **Разработка через тестирование (англ. *test-driven development, TDD*)** — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

Подробный текст лекции

0. Основы тестирования

Тестирование кода является неотъемлемой частью больших проектов. Мало кто пишет тесты для одноразовых задач в несколько строчек кода. Но если ваш проект планирует развиваться, тесты помогут экономить время при расширении функционала, внесении доработок в существующий код.

Обычно тестирование подразделяется на три категории:

1. Функциональное тестирование
 - Модульное (компонентное)
 - Интеграционное
 - Системное

- Регрессионное
 - Приемочное
 - Смоук
2. Тестирование производительности
- Тестирование отказоустойчивости
 - Нагрузочное
 - Объемное
 - Тестирование масштабируемости
3. Обслуживание (регресс и обслуживание)
- Регрессионное
 - Тестирование технического обслуживания

Список далеко не полный. Да и способов группировки тестирования больше много больше. В рамках данного курса и лекции не будут рассматриваться все возможные виды тестирования. Главная цель занятия - познакомить с тремя основными инструментами, позволяющими писать тесты для ваших Python проектов. При этом в рамках проекта обычно используется один из трёх рассматриваемых вариантов, а не все разом. По традиции разберём возможные варианты на примерах кода с пояснениями.

1. Основы doctest

Инструмент doctest встроен в Python и не требует дополнительных манипуляций по установке и настройке. Как заявляют сами разработчики языка doctest можно использовать для следующих задач:

- Проверка актуальности строк документации модуля путем проверки того, что все интерактивные примеры по-прежнему работают в соответствии с документацией.
- Для регрессионного тестирования. Чтобы убедиться, что интерактивные примеры из тестового файла или тестового объекта работают должным образом.
- Позволяют написать учебную документацию для пакета, обильно иллюстрированную примерами ввода-вывода. В зависимости от того, что выделено — примеры или пояснительный текст, это может быть что-то вроде “грамотного тестирования” или “исполняемой документации”.

Проверка примеров в документации, регрессионное тестирование

Как вы помните из прошлых лекций, тройные двойные кавычки сразу после заголовка класса, функции или метода превращают текст внутри в строку документации соответствующего объекта. Например так может выглядеть простейшая (без оптимизации) функция, проверяющая является ли число простым или составным используя нахождение остатка от деления.

```
def is_prime(p: int) -> bool:
    """
        Checks the number P for simplicity using finding the
        remainder of the division in the range [2, P).
    """
    for i in range(2, p):
        if p % i == 0:
            return False
    return True

help(is_prime)
```

А теперь сохраним код в файле main.py и сделаем несколько запусков в терминале в режиме интерпретатора.

```
>>> from main import is_prime
Help on function is_prime in module main:
is_prime(p: int) -> bool
    Checks the number P for simplicity using finding the
    remainder of the division in the range [2, P).
>>> is_prime(42)
False
>>> is_prime(73)
True
```

1. Мы сразу вспомнили, что команда import запускает импортируемый файл. У нас сработал вызов справки, потому что мы не спрятали его в `__name__ == "__main__"`
2. Вызов функции в режиме интерпретатора позволяет получить ответ для любых значений.

Если перенести вызову и результаты из консоли в строку документации функции (класса, модуля), получим тесты doctest. В нашем случае можно сделать так:

```
def is_prime(p: int) -> bool:
```

```

"""
    Checks the number P for simplicity using finding the
    remainder of the division in the range [2, P).
    >>> is_prime(42)
    False
    >>> is_prime(73)
    True
"""

for i in range(2, p):
    if p % i == 0:
        return False
return True

if __name__ == '__main__':
    import doctest
    doctest.testmod()

```

В документацию добавлены строки вызова функции в режиме интерпретатора. Они начинаются с тройной стрелки и пробела. Сразу после идёт строка с ответом. В “нейм-мейн” спрятали импорт модуля doctest и вызов функции testmod для тестирования кода.



Важно! doctest запускает код и сравнивает возвращаемое значение в виде текста с текстом внутри строки документации. Если допустить опечатку, поставить лишние отступы или ещё как-то изменить текст ответа, тест будет провален.

При запуске файла ничего не происходит. По умолчанию тестирование не выводит информации, если тесты прошли успешно. Попросим добавить вывод результатов в любом случае. Для этого исправим последнюю строку на `doctest.testmod(verbose=True)`

Теперь перед нами подробный вывод того, что тесты пройдены успешно.

Разработка через тестирование, TDD

Разработка через тестирование (англ. test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое

изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.

В TDD выделяют следующие этапы:

1. Добавление теста
2. Запуск всех тестов: убедиться, что новые тесты не проходят
3. Написать код
4. Запуск всех тестов: убедиться, что все тесты проходят
5. Рефакторинг
6. Повторить цикл

Применим TDD на практике.

➤ Добавление теста

Попробуем добавить в нашу функцию тесты для проверки особых случаев, а именно:

- Число должно быть натуральным.
 - Если функция вызывается не с целым, будем возвращать ошибку типа.
 - А если число будет целым, но не натуральным, ошибку значения.
- Отдельно напишем тест для единицы - натурального целого числа, которое не может быть проверено на простоту.
- Предусмотреть предупреждение о возможно долгом поиске ответа, если на простоту проверяется число больше ста миллионов.
 - Сделаем тест для большого составного числа
 - Отдельно сделаем тест для большого простого числа

В результате написания тестов получим следующий код:

```
def is_prime(p: int) -> bool:
    """
        Checks the number P for simplicity using finding the
        remainder of the division in the range [2, P).
    """
    >>> is_prime(42)
    False
    >>> is_prime(73)
    True
    >>> is_prime(3.14)
    Traceback (most recent call last):
        ...
    TypeError: The number P must be an integer type
    >>> is_prime(-100)
    Traceback (most recent call last):
        ...
    ValueError: The number P must be greater than 1
```

```

>>> is_prime(1)
Traceback (most recent call last):
...
ValueError: The number P must be greater than 1
>>> is_prime(100_000_001)
    If the number P is prime, the check may take a long time.
Working...
False
>>> is_prime(100_000_007)
    If the number P is prime, the check may take a long time.
Working...
True
"""
for i in range(2, p):
    if p % i == 0:
        return False
return True

if __name__ == '__main__':
    import doctest
    doctest.testmod()

```

Обратите внимание на многоточия в тестах ошибок. Модуль doctest ориентируется на первую строку об ошибке. Это или Traceback (most recent call last): или Traceback (innermost last): Так как номера строк кода, пути имена могут меняться, середина вывода игнорируется. Её можно заменить на многоточие. Важна последняя строка, которая начинается с типа ошибки и последующего за ней текста.

➤ Запуск всех тестов: убедиться, что новые тесты не проходят

Запускаем файл с кодом даже не включая режим отображения verbose. И получаем огромный список ошибок заканчивающийся следующим итогом:

```

*****
*****
1 items had failures:
    5 of  7 in __main__.is_prime
***Test Failed*** 5 failures.

```

Логично провалить 5 новых тестов, ведь мы пока не писали код, который позволит их пройти.



Важно! Так как doctest сравнивает текст, вызов ошибки, например через raise и печать аналогичного текста через print() будут восприниматься одинаково.

➤ Написать код

Самое время написать несколько строчек кода внутри функции is_prime(). На выходе получим следующий файл:

```
def is_prime(p: int) -> bool:
    """
        Checks the number P for simplicity using finding the
        remainder of the division in the range [2, P).
    """
    >>> is_prime(42)
    False
    >>> is_prime(73)
    True
    >>> is_prime(3.14)
    Traceback (most recent call last):
        ...
    TypeError: The number P must be an integer type
    >>> is_prime(-100)
    Traceback (most recent call last):
        ...
    ValueError: The number P must be greater than 1
    >>> is_prime(1)
    Traceback (most recent call last):
        ...
    ValueError: The number P must be greater than 1
    >>> is_prime(100_000_001)
    If the number P is prime, the check may take a long time.
    Working...
    False
    >>> is_prime(100_000_007)
    If the number P is prime, the check may take a long time.
    Working...
    True
    """
    if not isinstance(p, int):
        raise TypeError('The number P must be an integer type')
    elif p < 2:
        raise ValueError('The number P must be greater than 1')
    elif p > 100_000_000:
        print('If the number P is prime, the check may take a
        long time. Working...')
    for i in range(2, p):
```

```

        if p % i == 0:
            return False
        return True

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)

```

Понадобилось написать шесть строк кода, три проверки.

1. Убедиться что число целое, иначе вызвать ошибку типа.
2. Убедиться, что число не меньше двух, иначе вызвать ошибку значения. Тут мы проходим сразу два теста из пяти добавленных в техзадании.
3. Убедиться, что число меньше ста миллионов, иначе вывести предупреждение. Снова закрываем два теста одной проверкой

➤ Запуск всех тестов: убедиться, что все тесты проходят

Запуск кода без режима `verbose` ничего не выводит. Код успешно проходит тесты. С режимом отображения увидим успешное прохождение всех семи тестов:

```

...
7 passed and 0 failed.
Test passed.

```

Ура! Мы успешно справились с поставленной задачей.

Проверка исполняемой документации

Проект по созданию модуля работы с простыми числами оказался успешным. Мы сохранили код в `prime.py` и активно расширяем его возможности. Самое время написать файл документации к модулю. Создаём `prime.md` — текстовый файл, поддерживающий разметку Markdown.

```

Документация к модулю работы с простыми числами
===

```

```

Описание функции is_prime()
---

```

```

Для проверки числа на простоту используйте функцию is_prime

```

модуля prime.

Импортируйте её в свой код.

```
>>> from prime import is_prime
```

Теперь можно проверять числа на простоту.

```
>>> is_prime(2)
True
```

Функция использует проверку остатка от деления и может долго возвращать результат для больших простых чисел.

Вы получите предупреждение, это нормально. Просто подождите.

```
>>> is_prime(1000000007)
If the number P is prime, the check may take a long time.
Working...
True
```

Приятного использования нашего модуля ;-)



Важно! Обычно документацию пишут на английском языке. В учебном примере специально был выбран русский как основной язык на учебной платформе. Вы всегда можете использовать современные online переводчики для получения нужного языка.

В документации есть несколько примеров, имитирующих работу в режиме интерпретатора. Убедимся, что они рабочие, написав в отдельном файле пару строк кода.

```
import doctest

doctest.testfile('prime.md', verbose=True)
```

Функция testfile построчно читает переданный ей файл и если встречается примеры выполнения кода, тестирует их работоспособность. Обратите внимание, что мы указываем на необходимость импорта функции в самом начале документации. И после вызова оставили пустую строку. doctest сделает импорт и не будет ожидать ничего в ответ. Да, это строчка будет воспринята как тест. И без него все последующие примеры провалятся. Ведь режим интерпретатора работает последовательно.

Запуск тестов из командной строки

Прежде чем завершить введение работу с модулем doctest пара примеров запуска из терминала.



Важно! Не путайте терминал (консоль, командную строку) операционной системы и консольный режим работы интерпретатора Python.

```
PS C:\Users\PycharmProjects> python -m doctest .\prime.py
PS C:\Users\PycharmProjects> python -m doctest .\prime.py -v
PS C:\Users\PycharmProjects> python -m doctest .\prime.md
PS C:\Users\PycharmProjects> python -m doctest .\prime.md -v
```

Вызываем интерпретатор python и в качестве модуля указываем doctest. Далее передаём путь до файла, который хотим тестировать. Если файл имеет расширение py, запускается функция testmod (строки 1 и 2). А если у файла другое расширение, предполагается что это исполняемая документация и запускается функция testfile (строки 3 и 4). Дополнительный ключ -v включает режим подробного вывода результатов тестирования.



Внимание! Пример кода сделан в терминале PowerShell. В зависимости от используемого вами терминала строка приглашения может быть другой. Но текст команд одинаков для любой ОС и любого терминала.

Задание

Перед вами несколько строк doctest. Напишите что должна делать программа, чтобы пройти тесты. У вас 3 минуты.

```
"""
>>> say('Hello')
Hello Hello
>>> say('Hi', 5)
Hi Hi Hi Hi Hi
>>> say('cat', 3, '(=^.^=) ')
cat(=^.^=) cat(=^.^=) cat
"""
```

2. Основы тестирования с unittest

Рассмотрим более мощный по функциональности инструмент тестирования из коробки. Модуль unittest входит в стандартную библиотеку Python и не требует дополнительной установки. Более того, unittest называют фреймворком, а не просто модулем.

Среда unittest модульного тестирования изначально была вдохновлена JUnit и имеет тот же вкус, что и основные среды модульного тестирования на других языках. Он поддерживает автоматизацию тестирования, совместное использование кода установки и завершения тестов, объединение тестов в коллекции и независимость тестов от структуры отчетности.



Внимание! Вдохновение JUnit сказалось на стиле фреймворка, а именно на использовании camelCase для имён, вместо привычного для Python разработчика стиля snake_case.

Общие моменты работы с unittest

Рассмотрим некоторые общие моменты работы с unittest на примере следующего кода.

```
import unittest

class TestCaseName(unittest.TestCase):

    def test_method(self):
        self.assertEqual(2 * 2, 5, msg='Видимо и в этой вселенной
не работает :-(')

if __name__ == '__main__':
    unittest.main()
```

Для хранения тестов рекомендуется создавать отдельный файл с тестами или папку tests, если файлов с тестами будет много. Смешивать в одном файле исполняемый код и тесты не рекомендуется.

В файле с тестом импортируем модуль unittest и создаём класс для тестирования - test case. Такой класс должен наследоваться от TestCase.

Внутри класса создаём методы, имена которых должны начинаться со слова test. Таких методов внутри класса может быть несколько.

По наследованию от класса TestCase и именам методов unittest понимает, что перед ним тесты, которые необходимо запустить.

Для проверки используем утверждения - “асерты”. В приведённом примере assertEquals принимает два аргумента: $2 * 2$ и 5. Тест утверждает, что они равны. А если значения не равны, будет поднято исключение AssertionError с текстом, который передали в ключевом параметре msg.



Внимание! Реальные тесты не должны содержать неверные утверждения, подобные “дважды два равно пяти”.

Для запуска тестов вызываем функцию main(). Она проанализирует файл, соберёт тестовые кейсы, запустит и сообщит результаты проверки.



Внимание! Команда для запуска тестов из командной строки выглядит аналогично запуску doctest

```
$ python3 -m unittest tests.py -v
```

Сравнение тестов doctest и unittest

Возьмём уже знакомую функцию проверки числа на простоту и реализуем написанные ранее в doctest тесты используя unittest.

Файл prime.py без тестов doctest

```
def is_prime(p: int) -> bool:
    if not isinstance(p, int):
        raise TypeError('The number P must be an integer type')
    elif p < 2:
        raise ValueError('The number P must be greater than one')
    elif p > 100_000_000:
        print('If the number P is prime, the check may take a
```

```

long time. Working...')
    for i in range(2, p):
        if p % i == 0:
            return False
    return True

```

Ничего нового в коде функции нет.

А так будет выглядеть файл test_prime.py

```

import io
import unittest
from unittest.mock import patch

from prime import is_prime

class TestPrime(unittest.TestCase):

    def test_is_prime(self):
        self.assertFalse(is_prime(42))
        self.assertTrue(is_prime(73))

    def test_type(self):
        self.assertRaises(TypeError, is_prime, 3.14)

    def test_value(self):
        with self.assertRaises(ValueError):
            is_prime(-100)
            is_prime(1)

    @patch('sys.stdout', new_callable=io.StringIO)
    def test_warning_false(self, mock_stdout):
        self.assertFalse(is_prime(100_000_001))
        self.assertEqual(mock_stdout.getvalue(),
                          'If the number P is prime, the check may
take a long time. Working...\n')

    @patch('sys.stdout', new_callable=io.StringIO)
    def test_warning_true(self, mock_stdout):
        self.assertTrue(is_prime(100_000_007))
        self.assertEqual(mock_stdout.getvalue(),
                          'If the number P is prime, the check may
take a long time. Working...\n')

if __name__ == '__main__':
    unittest.main()

```

Разберём каждый из тестов внутри класса:

➤ **Кейс test_is_prime**

Проверяем базовую работу функции. Утверждение `assertFalse` ожидает получить ложь в качестве аргумента. В нашем случае в качестве результата вызова функции. Аналогично `assertTrue` ожидает получить истину.

➤ **Кейс test_type**

Утверждение `assertRaises` ожидает ошибку типа (аргумент один) если вызвать функцию `is_prime` (аргумент два) и передать ей число 3.14 (аргумент три).

➤ **Кейс test_value**

Используем менеджер контекста для утверждения ошибки и внутри контекста дважды запускаем функцию. `assertRaises` во всех случаях будет ожидать ошибку значения

➤ **Кейсы test_warning_false и test_warning_true**



Внимание! Оба примера выходят за рамки основ `unittest`. Это скорее пример на будущее для самых любознательных.

Используя декоратор `patch` из модуля `mock` перенаправляем стандартный поток вывода `sys.stdout` обращаясь к `StringIO` модуля ввода-вывода `io`. Результат попадает в параметр `mock_stdout`. Внутри метода делаем стандартную проверку на ложь или истину для большого числа. А далее проверяем, что стандартный вывод получил значение, совпадающее с ожидаемым текстом предупреждения.



Внимание! Разбор `Mock` объектов выходит за рамки лекции. Самые любознательные могут обратиться к стандартной документации языка.
<https://docs.python.org/3.11/library/unittest.mock.html>

Запуск тестов `doctest` из `unittest`

А что если тесты уже написаны в `doctest`? В этом случае можно создать функцию `test_loader` и добавить тесты `doctest` в перечень для тестирования. Изучите пример.

```

import doctest
import unittest

import prime

def load_tests(loader, tests, ignore):
    tests.addTests(doctest.DocTestSuite(prime))
    tests.addTests(doctest.DocFileSuite('prime.md'))
    return tests

if __name__ == '__main__':
    unittest.main()

```

Объект tests используя метод addTests добавляет импортированный модуль prime. Для этого используется класс DocTestSuite из модуля doctest. А если необходимо тестировать документацию, используется класс DocFileSuite. Теперь функция unittest.main соберёт написанные ранее тесты doctest и запустит их.

Подготовка теста и сворачивание работ

Иногда бывает необходимо выполнить какие-то действия до начала тестирования, развернуть тестируемую среду. А после завершения теста наоборот, убрать лишнее. Для этих целей в unittest есть зарезервированные имена методов setUp и tearDown. Часто их называют фикстурами.

➤ Метод setUp

Когда внутри класса есть несколько тестовых методов, вызов метода setUp происходит перед каждым вызовом теста.

```

import unittest

class TestSample(unittest.TestCase):

    def setUp(self) -> None:
        self.data = [2, 3, 5, 7]
        print('Выполнил setUp') # Только для демонстрации работы
        метода

    def test_append(self):

```

```

        self.data.append(11)
        self.assertEqual(self.data, [2, 3, 5, 7, 11])

    def test_remove(self):
        self.data.remove(5)
        self.assertEqual(self.data, [2, 3, 7])

    def test_pop(self):
        self.data.pop()
        self.assertEqual(self.data, [2, 3, 5])

if __name__ == '__main__':
    unittest.main()

```

В примере трижды создаётся список на четыре элемента. Каждый из тестов ожидает, что будет работать с числами 2, 3, 5, 7 и никак не учитывает результаты работы других тестов. Подобный подход удобен, когда надо прогнать большое количество тестов на одном и том же наборе данных.

➤ Метод `tearDown`

Метод `tearDown` будет вызван после успешного выполнения метода `setUp` и в случае если тест отработал успешно, и если он провалился.

```

import unittest

class TestSample(unittest.TestCase):

    def setUp(self) -> None:
        with open('top_secret.txt', 'w', encoding='utf-8') as f:
            for i in range(10):
                f.write(f'{i:05}\n')

    def test_line(self):
        with open('top_secret.txt', 'r', encoding='utf-8') as f:
            for i, line in enumerate(f, start=1):
                pass
        self.assertEqual(i, 10)

    def test_first(self):
        with open('top_secret.txt', 'r', encoding='utf-8') as f:
            first = f.read(5)
            self.assertEqual(first, '00000')

    def tearDown(self) -> None:

```

```
from pathlib import Path
Path('top_secret.txt').unlink()

if __name__ == '__main__':
    unittest.main()
```

В примере метод setUp создаёт перед каждым тестом файл со строками чисел. Два теста работают с этим файлом. И после каждого происходит удаление файла из tearDown метода.

Даже если провалить тест, файл будет удалён.

Перечень доступных утверждений assert

В списке ниже приведены доступные в unittest утверждения и пояснения о том что именно они проверяют.

- assertEquals(a, b) - a == b
- assertNotEqual(a, b) - a != b
- assertTrue(x) - bool(x) is True
- assertFalse(x) - bool(x) is False
- assertIs(a, b) - a is b
- assertIsNot(a, b) - a is not b
- assertIsNone(x) - x is None
- assertIsNotNone(x) - x is not None
- assertIn(a, b) - a in b
- assertNotIn(a, b) - a not in b
- isinstance(a, b) - isinstance(a, b)
- assertNotIsInstance(a, b) - not isinstance(a, b)
- assertRaises(exc, fun, *args, **kwargs) - функция fun(*args, **kwargs) поднимает исключение exc
- assertRaisesRegex(exc, r, fun, *args, **kwargs) - функция fun(*args, **kwargs) поднимает исключение exc и сообщение совпадает с регулярным выражением r
- assertWarns(warn, fun, *args, **kwargs) - функция fun(*args, **kwargs) поднимает предупреждение warn
- assertWarnsRegex(warn, r, fun, *args, **kwargs) - функция fun(*args, **kwargs) поднимает предупреждение warn и сообщение совпадает с регулярным выражением r
- assertLogs(logger, level) - блок with записывает логи в logger с уровнем level

- `assertNoLogs(logger, level)` - блок with не записывает логи в logger с уровнем level
- `assertAlmostEqual(a, b)` - `round(a-b, 7) == 0`
- `assertNotAlmostEqual(a, b)` - `round(a-b, 7) != 0`
- `assertGreater(a, b)` - `a > b`
- `assertGreaterEqual(a, b)` - `a >= b`
- `assertLess(a, b)` - `a < b`
- `assertLessEqual(a, b)` - `a <= b`
- `assertRegex(s, r)` - `r.search(s)`
- `assertNotRegex(s, r)` - `not r.search(s)`
- `assertCountEqual(a, b)` - a и b содержат одни и те же элементы в одинаковом количестве независимо от их порядка в коллекциях

Как вы видите перечень допустимых проверок достаточно обширный, чтобы удовлетворить практически любые запросы по написанию тестов.

Задание

Перед вами несколько строк кода. Напишите что должна делать программа, чтобы пройти тесты. У вас 3 минуты.

```
import unittest
from main import func

class TestSample(unittest.TestCase):

    def setUp(self) -> None:
        self.data = {'one': 1, 'two': 2, 'three': 3, 'four': 4}

    def test_step_1(self):
        self.assertEqual(func(self.data), 4)

    def test_step_2(self):
        self.assertEqual(func(self.data, first=False), 2)

if __name__ == '__main__':
    unittest.main()
```

3. Основы тестирования с pytest

Финальный модуль для создания тестов — `pytest`. Он не входит в стандартную библиотеку Python, поэтому должен быть установлен перед использованием.

```
pip install pytest
```



Важно! Версия Python должна быть 3.7 или выше.

Команда `assert`

В Python есть зарезервированное слово `assert`. После работы с `unittest` вы догадываетесь о её назначении. `assert` делает утверждение. Если оно возвращает истину, программа продолжает работать. А если утверждение ложно, поднимается ошибка `AssertionError`.

Для простоты можно представить `assert` как особую конструкцию `if`.

- “асерт”

```
assert утверждение, "Утверждение не подтвердилось"
```

- “иф”

```
if утверждение:
    pass
else:
    raise AssertionError("Утверждение не подтвердилось")
```

Модуль `pytest` формирует свою работу вокруг встроенной команды `assert`. Но в отличие от примера с `if` даёт подробную информацию об ошибках, если тесты не проходят.

Общие моменты работы с pytest

Рассмотрим простой пример тестирования работы функции, которая складывает два числа.

```
import pytest

def sum_two_num(a, b):
    return a + b
    # return f'{a}{b}'

def test_sum():
    assert sum_two_num(2, 3) == 5, 'Математика покинула чат'

if __name__ == '__main__':
    pytest.main()
```

Импорт модуля нужен только для запуска тестов из файла. Для создания простейших тестов модуль pytest не нужен.

Функция `sum_two_num` наш подопытный. Она принимает пару числе и возвращает их сумму.

Для создания кейса просто определяем функцию, которая начинается со слова `test`. Внутри используем `assert` для проверки утверждения. В простейшем случае это сравнение вызова функции с ожидаемым результатом. Более сложные ассерты рассмотрим далее. Дополнительно можно указать сообщение, которое будет выведено в случае провала теста.

Количество функций в файле может быть любым. pytest найдёт и запустит их все на основе сопоставления имён. При этом превращать функции в методы класса как в `unittest` не нужно. А если очень хочется объединить кейсы внутри класса, создайте класс начинающийся с `Test`.

Чтобы запустить тест из файла, вызываем функцию `main` из модуля `pytest`.



Внимание! Команда для запуска тестов из командной строки выглядит аналогично запуску `doctest` и `unittest`. Ключ с одиночным или двойным `v` указывает на уровень детализации. Кроме того можно вызывать `pytest` напрямую.

```
$ python3 -m pytest tests.py -vv
$ pytest tests_pt.py
```

Сравнение тестов pytest с doctest и unittest

Ещё раз возьмём функцию проверки числа на простоту и реализуем написанные ранее тесты используя pytest.

Файл prime.py не изменился

```
def is_prime(p: int) -> bool:
    if not isinstance(p, int):
        raise TypeError('The number P must be an integer type')
    elif p < 2:
        raise ValueError('The number P must be greater than one')
    elif p > 100_000_000:
        print('If the number P is prime, the check may take a
long time. Working...')
    for i in range(2, p):
        if p % i == 0:
            return False
    return True
```

Файл test_prime_pt.py с кейсами pytest

```
import pytest

from prime import is_prime

def test_is_prime():
    assert not is_prime(42), '42 - составное число'
    assert is_prime(73), '73 - простое число'

def test_type():
    with pytest.raises(TypeError):
        is_prime(3.14)

def test_value():
    with pytest.raises(ValueError):
        is_prime(-100)

def test_value_with_text():
```

```

    with pytest.raises(ValueError, match=r'The number P must be
greater than 1'):
        is_prime(1)

def test_warning_false(capfd):
    is_prime(100_000_001)
    captured = capfd.readouterr()
    assert captured.out == 'If the number P is prime, the check
may take a long time. Working...\n'

def test_warning_true(capfd):
    is_prime(100_000_007)
    captured = capfd.readouterr()
    assert captured.out == 'If the number P is prime, the check
may take a long time. Working...\n'

if __name__ == '__main__':
    pytest.main(['-v'])

```

Начало с импортом и конец с запуском тестов стандартные для Python. Разберём каждый из кейсов.

➤ Кейс test_is_prime

Проверяем базовую работу функции. Утверждение `assert not` ожидает получить ложь в качестве результата вызова функции. Второй `assert` ожидает получить истину.



Важно! Если первая строка провалит тест, второй `assert` не будет вызван для проверки. Обычно внутри кейса пишут одно утверждения. В нашем случае можно разделить проверки на два отдельных кейса.

➤ Кейс test_type

Используем менеджер контекста `pytest.raises` который ожидает получить ошибку `TypeError` при вызове `is_prime` с вещественным числом в качестве аргумента. Наличие ошибки проходит тест, а её отсутствие - роняет. Строка, которая должна поднять ошибку - последняя строка внутри менеджера контекста. Дальнейший код не будет выполняться.

➤ Кейс `test_value`

Тест работает аналогично проверки типа, но мы указали другую ошибку в менеджере и передали другое значение в функцию.

➤ Кейс `test_value_with_text`

Более сложный подход к тестированию. Помимо ошибки в параметр `match` передаётся регулярное выражение. Если оно совпадёт с текстом ошибки, тест будет пройден.

➤ Кейсы `test_warning_false` и `test_warning_true`

Внимание! Оба примера выходят за рамки основ `pytest`. Это скорее пример на будущее для самых любознательных.

Кейс получает фикстуру `capfd` в качестве аргумента. `capfd` (capture file descriptors) является одной из встроенных в `pytest` фикстур, которая позволяет перехватывать потоки вывода и ошибок. Внутри кейса вызываем тестируемую функцию. Далее используем метод `readouterr()` для получения потоков в переменную `captured`. В финале сравниваем результат `captured.out` (потока вывода) с ожидаемым текстом сообщения.

Запуск тестов `doctest` и `unittest`

Для запуска тестов, написанных другими инструментами можно воспользоваться следующими командами в консоли ОС:

```
$ pytest --doctest-modules prime.py -v
$ pytest tests_ut.py
```

В случае с `doctest` необходимо указать флаг `--doctest-modules`. Для `unittest` ничего указывать не надо. Практически все кейсы из `unittest` можно запускать в `pytest`. Модуль понимает синтаксис, способен собрать тесты из классов и проверить их.

Фикстуры pytest как замены unittest setUp и tearDown

Если вам необходимо выполнить однотипные действия для подготовки нескольких тестов, можно создать собственные фикстуры. Рассмотрим простой пример из главы о unittest.

```
import pytest

@pytest.fixture
def data():
    return [2, 3, 5, 7]

def test_append(data):
    data.append(11)
    assert data == [2, 3, 5, 7, 11]

def test_remove(data):
    data.remove(5)
    assert data == [2, 3, 7]

def test_pop(data):
    data.pop()
    assert data == [2, 3, 5]

if __name__ == '__main__':
    pytest.main(['-v'])
```

Функция data превращается в фикстуру добавлением декоратора @pytest.fixture. Чтобы использовать фикстуру внутри кейса, необходимо передать её в качестве аргумента. Выбранный разработчиком pythtest подход к фикстурам удобен тем, что позволяет любые вариации с кейсами.

- можно иметь множество фикстур и разные кейсы могут использовать разные фикстуры.
- в кейс можно передать любое количество фикстур.
- фикстура может принимать в качестве аргумента другую фикстуру.

Ещё немного о фикстурах

Рассмотрим пример посложнее.

```
import pytest

@pytest.fixture
def get_file(tmp_path):
    f_name = tmp_path / 'test_file.txt'
    print(f'Создаю файл {f_name}') # принтим в учебных целях
    with open(f_name, 'w+', encoding='utf-8') as f:
        yield f
    print(f'Закрываю файл {f_name}') # принтим в учебных целях

@pytest.fixture
def set_num(get_file):
    print(f'Заполняю файл {get_file.name} цифрами') # принтим в
    учебных целях
    for i in range(10):
        get_file.write(f'{i:05}')
    get_file.seek(0)

@pytest.fixture
def set_char(get_file):
    print(f'Заполняю файл {get_file.name} буквами') # принтим в
    учебных целях
    for i in range(65, 91):
        get_file.write(f'{chr(i)}')
    get_file.seek(0)
    return get_file

def test_first_num(get_file, set_num):
    first = get_file.read(5)
    assert first == '00000'

def test_first_char(set_char):
    first = set_char.read(5)
    assert first == 'ABCD' # специально провалим тест

if __name__ == '__main__':
    pytest.main(['-v'])
```


Кейсов всего два: `test_first_num` и `test_first_chr`. И каждый из них использует свои фикстуры. Но давайте обо всём сверху вниз.

➤ Фикстура `get_file`

Фикстура принимает на вход аргумент `tmp_path`. Это встроенная фикстура, которая возвращает временный путь - объект `pathlib.Path`. При использовании выводим не печать информацию о создании файла и о его удалении. Отследим когда срабатывает фикстура.

Внутри менеджера контекста создаём файл и через команду `yield` возвращаем указатель на него. Если бы мы использовали команду `return`, менеджер контекста вызвал бы `f.close()` после возврата указания и файл стал бы нечитаемым.

Используя `yield` мы превратили функцию в генератор. Теперь внутри фикстуры есть “сетап” создающий файл и “тирдаун”, закрывающий его после использования.

Внимание! Мы явно не удаляем временные файлы. Фикстура `tmp_path` сохраняет три последних временных каталога, удаляя старые при очередном запуске.

➤ Фикстура `set_num`

Используя файловый дескриптор `get_file` записываем строку из цифр и возвращаем указатель на начало файла. Фикстура ничего не возвращает.

➤ Фикстура `set_char`

Снова используем файловый дескриптор `get_file`, но получаем уже другой файл. Имя совпадает, но каталоги разные. Заполняем его буквами, сбрасываем позицию в ноль и возвращаем `get_file` - файловый дескриптор.

➤ Кейс `test_first_num`

Перед началом теста срабатывают фикстуры, создающие временный файл и заполняющие его цифрами. Далее обращаемся к `get_file` чтобы прочитать пять первых символов и сравниваем их со строкой текста.

➤ Кейс `test_first_char`

Кейс получает всего одну фикстуру `set_char`. Но так как она самостоятельно вызывает фикстуру `get_file` и возвращает её, мы можем обращаться к файловому дескриптору по имени `set_char`.

Рассмотренный пример даёт представление о гибкости кейсов `pytest`.

Субъективное мнение автора курса, но `pytest` является лучшим из трёх рассмотренных инструментов тестирования. Попробуйте все три, составьте своё.

Задание

Перед вами несколько строк кода. Напишите что должна делать программа, чтобы пройти тесты. У вас 3 минуты.

```
import pytest
from main import func

def test_1():
    assert func(4) == 0

def test_2():
    assert func(4, -4) == (1, 0)

def test_3():
    assert func(4, -10, -50) == (5, -2.5)

def test_4():
    assert func(1, 1, 1) is None

if __name__ == '__main__':
    pytest.main(['-v'])
```

Вывод

На этой лекции мы:

1. Разобрались с написанием тестов в Python
2. Изучили возможности doctest
3. Узнали о пакете для тестирования unittest
4. Разобрались с тестированием через pytest

Краткий анонс следующей лекции

1. Узнаем о составе стандартной библиотеки Python
2. Разберёмся в настройках логирования
3. Изучим работу с датой и временем
4. Узнаем ещё пару полезных структур данных
5. Изучим способы парсинга аргументов при запуске скрипта с параметрами

Домашнее задание

Возьмите 1-3 задачи из прошлых занятий. Напишите к ним тесты. Попробуйте написать одинаковые тесты в трёх инструментах. Так у вас будет возможность сравнить их.

Оглавление

На этой лекции мы	2
Дополнительные материалы к лекции	3
Краткая выжимка, о чём говорилось в предыдущей лекции	3
Подробный текст лекции	
1. Обзор библиотеки целиком	3
2. Модель logging	5
Уровни логирования	5
Базовые регистраторы	6
Подробнее о basicConfig	8
Уровень логирования	8
Файл журнала	8
Формат сохранения события	9
Задание	10
3. Модуль datetime	10
Создаём дату и время	11
Разница времени	11
Математика с датами	12
Доступ к свойствам	13
Другие методы работы с датой и временем	14
Задание	16
4. Пара полезных структур данных	
Модуль collections	16
Фабричная функция namedtuple	16
Модуль array	19
Задание	20
5. Модуль argparse	21
Ключ --help или -h	22
Запуск с неверными аргументами	23
Создаём парсер, ArgumentParser	23
Выгружаем результаты, parse_args	23
Добавляем аргументы, add_argument	24
Необязательные аргументы и значения по умолчанию	25
Параметр action для аргумента	26
Вывод	27
Домашнее задание	27

На этой лекции мы

1. Узнаем о составе стандартной библиотеки Python.
2. Разберёмся в настройках логирования
3. Изучим работу с датой и временем
4. Узнаем ещё пару полезных структур данных
5. Изучим способы парсинга аргументов при запуске скрипта с параметрами

Дополнительные материалы к лекции

Стандартная библиотека Python <https://docs.python.org/3.11/library>

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с написанием тестов в Python
2. Изучили возможности doctest
3. Узнали о пакете для тестирования unittest
4. Разобрались с тестированием через pytest

Подробный текст лекции

1. Обзор библиотеки целиком

Мы уже упоминали стандартную библиотеку Python на курсе. Даже использовали некоторые пакеты из неё. Вспомним, что же это такое.

Стандартная библиотека Python (The Python Standard Library) — распространяется вместе с интерпретатором Python. Следовательно для использования пакетов и модулей из неё достаточно написать `import name`.

Предлагаю для краткости называть Стандартную библиотеку Python просто библиотека в рамках этой лекции.

Библиотека очень обширна и позволяет решать огромный спектр задач. Внутри есть модули написанные на языке C. Благодаря ним у разработчиков есть лёгкий доступ к системным функциям. Другие модули написаны на Python и предлагают готовые решения для повседневных задач программирования.

Библиотека содержит:

- Встроенные функции
- Встроенные константы
- Встроенные типы
- Встроенные исключения
- Услуги обработки текста
- Службы двоичных данных
- Типы данных
- Числовые и математические модули
- Модули функционального программирования
- Доступ к файлам и каталогам
- Сохранение данных
- Сжатие данных и архивирование
- Форматы файлов
- Криптографические услуги
- Общие службы операционной системы
- Параллельное выполнение
- Сеть и межпроцессное взаимодействие
- Обработка данных в Интернете
- Инструменты обработки структурированной разметки
- Интернет-протоколы и поддержка
- Мультимедийные услуги
- Интернационализация
- Программные фреймворки
- Графические пользовательские интерфейсы с Tk
- Инструменты разработки

- Отладка и профилирование
- Упаковка и распространение программного обеспечения
- Службы выполнения Python
- Пользовательские интерпретаторы Python
- Импорт модулей
- Языковые службы Python
- Специальные службы MS Windows
- Специальные службы Unix
- Замененные (устаревшие) модули

И под каждым пунктом кроется несколько пакетов для решения соответствующих теме задач. Вы можете перейти по ссылке и увидеть детальный состав библиотеки самостоятельно <https://docs.python.org/3.11/library/index.html>.

Далее в рамках лекции рассмотрим несколько полезных модулей.

2. Модель logging

Модуль logging позволяет регистрировать события в приложениях. Для этого разработчику предоставляется гибкая система классов и функций. Традиционно начнём с простого примера.

```
import logging

logging.info('Немного информации')
logging.error('Поймали ошибку')
```

В результате получим строку:

```
ERROR:root:Поймали ошибку
```

Мы не увидели в консоли первое сообщение. По умолчанию модуль не реагирует на информационные сообщения. При этом логгер сообщил об ошибке от имени корневого регистратора — root.

Уровни логирования

По умолчанию логгер имеет следующие уровни журналирования

- NOTSET, 0 — уровень не установлен. Регистрируются все события.
- DEBUG, 10 — подробная информация, обычно представляющая интерес только при диагностике проблем.
- INFO, 20 — подтверждение того, что все работает так, как ожидалось.
- WARNING, 30 — указание на то, что произошло что-то неожиданное, или указание на какую-то проблему в ближайшем будущем (например, «недостаточно места на диске»). Программное обеспечение по-прежнему работает, как ожидалось.
- ERROR, 40 — из-за более серьезной проблемы программное обеспечение не может выполнять некоторые функции.
- CRITICAL, 50 — серьезная ошибка, указывающая на то, что сама программа не может продолжать работу.

Уровень NOTSET используется как значение по умолчанию при создании обработчиков событий. Все остальные уровни имеют одноимённые функции — регистраторы, которые позволяют зафиксировать события заданного уровня.

Кроме того уровни имеют числовой эквивалент от 0 до 50. Он нужен при создании своих уровней логирования, чтобы определить место обработчика относительно базовых.



Важно! На протяжении курса мы использовали функцию `print()` чтобы посмотреть отладочную информацию в том или ином коде. Обычно, но не всегда, рядом с таким принтом составлялся комментарий, что этот вывод для учебных целей, а не для реальных проектов. Правильно было бы использовать логирование уровня `debug` или `info` вместо функции `print`.

Базовые регистраторы

Имена функций для регистрации событий совпадают с названиями констант для определения уровня логирования. Перечислим их в примере кода.

```
import logging

logging.basicConfig(level=logging.NOTSET)
logging.debug('Очень подробная отладочная информация. Заменяем множество "принтов"')
logging.info('Немного информации о работе кода')
```



```
logging.warning('Внимание! Надвигается буря!')
logging.error('Поймали ошибку. Дальше только неизвестность')
logging.critical('На этом всё')
```

Разработчик сам выбирает какой уровень использовать для регистрации того или иного события в его коде.

Обычно в коде не используют прямое обращение к регистраторам через имя модуля. В официальной документации указано, что работать с регистраторами напрямую запрещено. Необходимо использовать функцию уровня модуля `logging.getLogger(name)` для получения регистраторов. В таком случае пример выше должен выглядеть так:

```
import logging

logging.basicConfig(level=logging.NOTSET)
logger = logging.getLogger(__name__)

logger.debug('Очень подробная отладочная информация. Заменяем
множество "принтов"')
logger.info('Немного информации о работе кода')
logger.warning('Внимание! Надвигается буря!')
logger.error('Поймали ошибку. Дальше только неизвестность')
logger.critical('На этом всё')
```

Напомним, что переменная `__name__` хранит имя модуля. Если мы запускаем модуль как исполняемый файл, в `__name__` хранится `__main__`. Если файл импортирован, `__name__` хранит имя файла.

А теперь рассмотрим ситуацию с несколькими файлами, когда мы работаем со сложным проектом.

Код основного файла:

```
import logging
from other import log_all

logging.basicConfig(level=logging.WARNING)
logger = logging.getLogger('Основной файл проекта')
logger.warning('Внимание! Используем вызов функции из другого
модуля')
log_all()
```

Код другого файла в проекте

```
import logging
```

```
logger = logging.getLogger(__name__)

def log_all():
    logger.debug('Очень подробная отладочная информация. Заменяем  
множество "принтов"')
    logger.info('Немного информации о работе кода')
    logger.warning('Внимание! Надвигается буря!')
    logger.error('Поймали ошибку. Дальше только неизвестность')
    logger.critical('На этом всё')
```

В основном коде определили уровень логирования - WARNING. Логер вывел сообщение и вместо root указал текст, переданный в функцию getLogger.

Далее мы вызываем импортированную функцию и...

В файле other так же импортирован модуль logging. Далее мы получаем регистратор с именем other, оно содержится в __name__. И не смотря на попытку использовать все уровни логирования, срабатывают только предупреждения и выше. Функция getLogger взяла конфигурацию basicConfig из основного файла проекта.

Подробнее о basicConfig

Функция basicConfig нужна для базовой настройки логгеров. Она должна быть вызвана в основном потоке раньше, чем будут вызывать логгеры любого уровня.

Функция принимает только ключевые параметры, так что важен не порядок, а имя.

➤ Уровень логирования

Как вы уже догадались ключевой параметр level принимает константу с числом для определения уровня логирования. Все значения меньше переданного будут игнорироваться регистраторами.

➤ Файл журнала

До этого момента информация логгеров выводилась в консоль. Но мы можем сохранять данные в файл, указав необходимые настройки в конфигурации.

```
import logging
from other import log_all
```

```
logging.basicConfig(filename='project.log.',          filemode='w',
encoding='utf-8', level=logging.INFO)
logger = logging.getLogger('Основной файл проекта')
logger.warning('Внимание! Используем вызов функции из другого
модуля')
log_all()
```

Параметры работы с файлами аналогичны функции `open()`.

- `filename` — путь и имя файла для сохранения журнала логирования.
- `filemode` — режим записи. Если передать 'w', каждый запуск будет удалять старые данные из журнала. Можно не указывать аргумента, тогда данные будут дописываться в конец файла
- `encoding` — кодировка для сохранения журнала
- `errors` — способ обработки ошибок при открытии файла. По умолчанию используется режим `backslashreplace`. Он заменяет искаженные данные управляющими последовательностями Python с обратной косой чертой.

➤ Формат сохранения события

Кроме того для базовой конфигурации используют параметр `format` для изменения стандартной строки логирования. Формат может быть задан в трёх стилях форматирования:

- “%” — старый стиль форматирования текста с использованием символа `%`. Это стиль по умолчанию для задания формата. Сохранён он для поддержки обратной совместимости код. Ведь логирование работа ещё в Python 1.5.2 созданном в прошлом тысячелетии.
- “{” — форматирование с использованием фигурных скобок. Метод `str.format`.
- “\$” — форматирование в стиле `string.Template` с использованием денежного символа для указания имени переменной.



Важно! Не путайте форматирование с использованием фигурных скобок в методе `str.format` с форматированием f-строк. Впрочем, они имеют много общего.

Рассмотрим стил с фигурными скобками как наиболее привычный современным разработчикам. Кстати, сам стиль надо указать при вызове конфигуратора в параметре `style`.

```
import logging
from other import log_all

FORMAT = '{levelname:<8} - {asctime}. В модуле "{name}" ' \
        'в строке {lineno:03d} функция "{funcName}()" ' \
```

```
        'в {created} секунд записала сообщение: {msg}'
logging.basicConfig(format=FORMAT, style='{', level=logging.INFO)
logger = logging.getLogger('main')
logger.warning('Внимание! Используем вызов функции из другого
модуля')
log_all()
```

Константа формат передаёт уровень логирования и время записи в читаемом формате. Далее указываем имя модуля и строку кода, откуда был вызван логгер. Если он находился в функции, то её имя и отдельно время срабатывания в секундах от 1 января 1970 года. В финале переменная `msg` выводит текст сообщения. Обратите внимание, что к некоторым значениям было применено форматирование.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
import logging

logging.basicConfig(
    filename="log/log.log",
    encoding='utf-8',
    format='{asctime} {levelname} {funcName}->{lineno}: {msg}',
    style='{',
    level=logging.WARNING
)
```

3. Модуль datetime

Ещё один полезный модуль — `datetime`. Как понятно из названия он необходим для обработки даты и времени.



Внимание! Вопрос работы с часовыми поясами специально опущен в этой главе. Но вы всегда можете обратиться к модулю `zoneinfo`, который появился в Python 3.9. Или установить внешний пакет, например `pytz`.



Важно! Прежде чем начать работать с датами, стоит помнить что модуль `datetime` представляет “вечный” Григорианский календарь. Т.е. он не учитывает другие календарные системы и считает что григорианский календарь всегда был и всегда будет.

Создаём дату и время

Следующий пример демонстрирует три основных типа данных модуля.

```
from datetime import time, date, datetime

d = date(year=2007, month=6, day=15)
t = time(hour=2, minute=14, second=0, microsecond=24)
dt = datetime(year=2007, month=6, day=15, hour=2, minute=14,
second=0, microsecond=24)
print(f'{d = }\t-\t{d}')
print(f'{t = }\t-\t{t}')
print(f'{dt = }\t-\t{dt}')
```

- `date` представляет дату в календаре от 1 января 1 года до 31 декабря 9999 года. Указывать год, месяц и число обязательно для создания объекта.
- `time` создаёт объект времени, который на вход принимает четыре необязательных параметра. Если любое из четырёх значений или даже все сразу отсутствуют, вместо него подставляется ноль
 - часы — от 0 до 23 часов
 - минуты — от 0 до 59 минут
 - секунды — от 0 до 59 секунд
 - микросекунды — от 0 до 999_999 микросекунд
- `datetime` является комбинацией даты и времени и при его создании действуют правила аналогичные для `date` и `time`.

Объекты являются неизменяемыми типами. Следовательно они возвращают хеш и могут использоваться как ключи словаря, элементы множества и т.д.

Разница времени

Ещё один важный тип данных — `timedelta`. Объект представляет из себя разницу во времени между различными датами и временными промежутками.

Простой пример создания временной разницы:

```
from datetime import timedelta

delta = timedelta(weeks=1, days=2, hours=3, minutes=4, seconds=5,
millisecons=6, microseconds=7)
print(f'{delta = }\t-\t{delta}')
```

Независимо от данных, которые используются для создания дельты, она хранит три значения:

- `days` — хранит переданные дни. Недели преобразуются в 7 дней
- `seconds` — хранит секунды. Минуты превращаются в 60 секунд, а часы в 3600 секунд
- `microseconds` — хранит микросекунды. Миллисекунды превращаются в 1000 микросекунд.

При выводе на печать получаем количество дней, часов, минут, секунд и микросекунд.

В отличии от трёх временных типов, дельты могут принимать на вход любые числовые значения. Например количество минут может быть больше 60. Кроме того дельта может быть отрицательной.

```
from datetime import timedelta

delta = timedelta(weeks=53, days=500, hours=73, minutes=101,
seconds=303, milliseconds=67890,
microseconds=1234567)
neg_delta = timedelta(days=-3, minutes=-42)
print(f'{delta = }\t-\t{delta}')
```

При переполнении любого из трёх значений, излишек преобразуется в следующее по старшинству. Главное, чтобы дни остались в диапазоне от минус миллиарда до плюс миллиарда.

Обратите внимание, что при отрицательных значениях с минусом хранятся только дни, а два других параметра остаются положительными и при хранении и при выводе информации.

Математика с датами

Разница во времени появляется при математических операциях с датами и временем. И эта же разница может использоваться для изменения дат. Например

```
from datetime import datetime, timedelta

date_1 = datetime(2012, 12, 21)
date_2 = datetime(2017, 8, 19)
delta = date_2 - date_1
print(f'{delta = }\t-\t{delta}')
```



```
birthday = datetime(1503, 12, 14)
dlt = timedelta(days=365.25 * 33)
new_date = birthday + dlt
print(f'{new_date = }\t-\t{new_date}')
```

В первом случае нашли разницу во времени между двумя событиями. А во втором вычислили дату тридцатитрёхлетия.

Доступ к свойствам

Каждый из объектов позволяет прочитать хранимые свойства обратившись к ним по имени через точечную нотацию.

```
from datetime import time, date, datetime, timedelta

d = date(year=2007, month=6, day=15)
t = time(hour=2, minute=14, microsecond=24)
dt = datetime(year=2007, month=6, day=15, hour=2, minute=14,
microsecond=24)
delta = timedelta(weeks=53, hours=73, minutes=101,
seconds=303, milliseconds=60)

print(f'{d.month}')
```

```
print(f'{t.second}')
```

```
print(f'{dt.hour}')
```

```
print(f'{delta.days}')
```

Даже если свойство явно не передано, но объект хранит его, мы можем получить доступ на чтение.

Изменить значение напрямую не получится. Всё же перед нами неизменяемые объекты. Но мы можем воспользоваться методом `replace` для создания копии со значениями текущего объекта. Изменения затронут только указанные параметры.



Внимание! `timedelta` не имеет метода `replace`.

```
from datetime import time, date, datetime, timedelta

t = time(hour=2, minute=14, microsecond=24)
dt = datetime(year=2007, month=6, day=15, hour=2, minute=14,
microsecond=24)

new_dt = dt.replace(year=2012)
one_more_hour = t.replace(t.hour + 1)
print(f'{new_dt}\n{one_more_hour}')
```

Другие методы работы с датой и временем

Мы можем использовать специальные методы для получения удобного вывода информации и наоборот, формировать объекты `datetime` из человекочитаемых строк.

Рассмотрим некоторые из них.

```
from datetime import datetime

dt = datetime(year=2007, month=6, day=15, hour=2, minute=14,
microsecond=24)

print(dt)
print(dt.timestamp())
print(dt.isoformat())
print(dt.weekday())
print(dt.strftime('Дата %d %B %Y. День недели %A. Время %H:%M:%S.
Это %W неделя и %j день года.'))
```

- `timestamp` — возвращаем время в секундах от начала времён. Под началом времён понимаем POSIX-время, т.е. полночь 1 января 1970 года. Программисты используют его как особую точку отсчёта, позволяющую

хранить время как целое (если нужна точность до секунды) или вещественное число. Подобный подход используется для хранения времени в БД и для манипуляций со временем.

- `isoformat` — выводит дату в формате, соответствующем стандарту ISO 8601. Это международный стандарт описывающий формат даты и времени и рекомендации по его использованию.
- `weekday` — позволяет получить день недели в виде целого числа, где 0 - понедельник, а 6 — воскресенье.
- `strftime` — выводит дату в соответствии с переданным в виде `str` форматом. Обычный текст выводится без изменения, а символ `%` указывает на следующий после него литер форматирования. Ознакомится со всеми можно по [ссылке](#).

А теперь несколько обратных методов, позволяющих создать объекты `datetime`.

```
from datetime import datetime

date_original = '2022-12-12 18:01:21.555470'
date_timestamp = 1181862840.000024
date_iso = '2007-06-15T02:14:00.000024'
date_text = 'Дата 15 June 2007. День недели Friday. Время 02:14:00. Это 24 неделя и 166 день года.'

original_date = datetime.fromisoformat(date_original)
timestamp_date = datetime.fromtimestamp(date_timestamp)
iso_date = datetime.fromisoformat(date_iso)
text_date = datetime.strptime(date_text, 'Дата %d %B %Y. День недели %A. Время %H:%M:%S. Это %W неделя и %j день года.')
print(original_date)
print(timestamp_date)
print(iso_date)
print(text_date)
```

- `fromisoformat` — метод анализирует строку текст и если она совпадает со стандартом ISO 8601, возвращает объект `datetime`. Стандартный вывод объектов на печать позволяет делать обратное преобразование этим методом.
- `fromtimestamp` — получаем объект даты из целого или вещественного числа. Само число — количество секунд после полуночи 1 января 1970.
- `strptime` — метод принимает на вход два параметра: строку для преобразования и строку с форматом. Если формат позволяет проанализировать строку, возвращается объект `datetime`.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
from datetime import datetime, timedelta

d = datetime.now()
td = timedelta(hours=1)
for i in range(24*7):
    if d.weekday() == 6:
        break
    else:
        d = d + td
print(i)
```

4. Пара полезных структур данных

Модуль collections

Модуль предоставляет доступ к встроенным в Python типам данных, но “спрятанным” от начинающего разработчика. Внутри хранится много интересных и полезных структур данных. А в `collections.abc` огромный набор абстрактных типов. Но сегодня мы рассмотрим функцию `namedtuple` из модуля.

Фабричная функция namedtuple

Функция `namedtuple` является фабрикой классов. Из названия следует, что она создаёт именованные кортежи. Рассмотрим простой пример, чтобы понять что это.

```
from collections import namedtuple
from datetime import datetime

User = namedtuple('User', ['first_name', 'last_name', 'birthday'])
u_1 = User('Исаак', 'Ньютон', datetime(1643, 1, 4))
print(u_1)
```

```
print(f'{type(User)}, {type(u_1)}')
```

Функция принимает пару обязательных значений:

1. Имя класса. Это строка, содержащее точно такое же имя как и переменная слева от знака равно.
2. Список строк или строка с пробелами в качестве разделителей. Имена из списка превращаются в свойства класса.

На выходе получаем класс, аналогичный созданному вручную классу class. При этом в классе помимо указанных в списке свойств автоматически создаются некоторые дандер методы. Например мы с лёгкостью распечатали экземпляр u_1, потому что он определил свой дандер __repr__.

Как и с экземплярами класса, мы можем получить доступ к свойствам используя точечную нотацию.

```
from collections import namedtuple
from datetime import datetime

User = namedtuple('User', ['first_name', 'last_name',
                           'birthday'])
u_1 = User('Исаак', 'Ньютон', datetime(1643, 1, 4))
print(u_1)
print(u_1.first_name, u_1.birthday.year)
```

Обратите внимание, что свойство день рождение является объектом datetime со своими свойствами. Доступ к ним мы получаем также через точечную нотацию

При создании класса можно дополнительно передать список значений по умолчанию. И если дефолтных значений меньше, чем свойств в классе, назначение происходит справа налево

```
import time
from collections import namedtuple
from datetime import datetime

User = namedtuple('User', ['first_name', 'last_name',
                           'birthday'], defaults=['Иванов', datetime.now()])
u_1 = User('Исаак')
print(f'{u_1.last_name}, {u_1.birthday.strftime("%H:%M:%S")}')
time.sleep(7)
u_2 = User('Галилей', 'Галилео')
print(f'{u_2.last_name}, {u_2.birthday.strftime("%H:%M:%S")}')
```

Составление списка с именами полей и значениями по умолчанию движется справа налево, поэтому в birthday попадает текущая дата, а фамилию по умолчанию -

Иванов. Значения подставляются только в том случае, когда экземпляр не передаёт свои, как и с обычными классами.



Внимание! Посмотрите на даты у каждого из экземпляров. Время совпадает до секунды несмотря на 7 секунд разницы в создании. Значения для birthday было вычислено один раз, в момент создания класса.

На самом деле ситуация с функциями неоднозначно. С одной стороны ничего не мешает присвоить экземпляру в качестве свойства созданную функцию. Но в отличие от классических классов, классы `namedtuple` рассчитаны на хранение свойств, а не методов.



Важно! Если вам нужен объект с методами, используйте классический ООП.

Как и в случае с неизменяемыми датами, экземпляры `namedtuple` также неизменны. Но если надо внести правку в какое-то поле, встроенный метод `_replace` создаст копию, заменив только указанные значения.

```
from collections import namedtuple

Point = namedtuple('Point', 'x y z', defaults=[0, 0, 0])
a = Point(2, 3, 4)
b = a._replace(z=0, x=a.x + 4)
print(b)
```

Экземпляры можно сортировать. Метод проверки “меньше” определяется для свойств автоматически

```
from collections import namedtuple

Point = namedtuple('Point', 'x y z', defaults=[0, 0, 0])
data = [Point(2, 3, 4), Point(10, -100, -500), Point(3, 7, 11),
        Point(2, 202, 1)]
print(sorted(data))
```

Как вы могли заметить при совпадении значений первого по счёту свойства происходит сравнение второго и т.д. пока не определится меньший элемент. И ещё одна интересная особенность `namedtuple`. Если все свойства являются объектами неизменяемого типа, экземпляр может быть ключом словаря, элементом множества и т.п.

```
from collections import namedtuple
```

```

Point = namedtuple('Point', 'x y z', defaults=[0, 0, 0])
d = {
    Point(2, 3, 4): 'first',
    Point(10, -100, -500): 'second',
    Point(3, 7, 11): 'last',
}
print(d)

mut_point = Point(2, [3, 4, 5], 6)
print(mut_point)
d.update({mut_point: 'bad_point'}) # TypeError: unhashable type:
'list'

```

Точка `mut_point` была создана и ошибки нет. `namedtuple` допускает изменяемые типы для свойств. Но такой экземпляр перестают быть хэшируемым. Как результат ошибка при добавлении точки в качестве ключа словаря.

И конечно же стоит упомянуть о таком плюсе `namedtuple` как экономия памяти. Экземпляры занимают в памяти столько же, сколько и обычные кортежи.

Модуль array

А вот ещё один модуль, предоставляющий доступ к типу данных массив. Как вы помните список `list` является массивом ссылок на объекты. А сами объекты хранятся отдельно, вне массива. `array` из модуля `array` является классическим массивом. Внутри него можно хранить целые или вещественные числа, а также символы Unicode.

```

from array import array, typecodes

byte_array = array('B', (1, 2, 3, 255))
print(byte_array)
print(typecodes)

```

Первый аргумент — строковой символ код типа. Буква указывает на то какие данные хранятся в массиве и выделяет нужное число байт под данные.

Вторым аргументом передают последовательность для помещения в массив.

Переменная `typecodes` выводит все допустимые коды типа:

- 'b' — целое со знаком, 1 байт
- 'B' — целое без знака, 1 байт

- 'u' — Юникод-символ в 2 или 4 байта
- 'h' — целое со знаком, 2 байта
- 'H' — целое без знака, 2 байта
- 'i' — целое со знаком, 4 байта
- 'I' — целое без знака, 4 байта
- 'l' — целое со знаком, 4 байта
- 'L' — целое без знака, 4 байта
- 'q' — целое со знаком, 8 байт
- 'Q' — целое без знака, 8 байт
- 'f' — вещественное обычной точности, 4 байта
- 'd' — вещественное двойной точности, 8 байт

Массивы поддерживают методы списка list, поэтому использование их интуитивно понятно. Привыкать надо лишь к указанию хранимого типа данных.

```
from array import array

long_array = array('l', [-6000, 800, 100500])
long_array.append(42)
print(long_array)
print(long_array[2])
print(long_array.pop())
```

При этом массив не позволит добавить значение, если оно выходит за пределы диапазона, заданного кодом типа при создании. Так же будет поднята ошибка при несоответствии типа.

```
from array import array

long_array = array('l', [-6000, 800, 100500])
long_array.append(2**32)    # OverflowError: Python int too large
                             # to convert to C long
long_array.append(3.14)    # TypeError: 'float' object cannot be
                             # interpreted as an integer
```

Массивы array являются более экономичной по памяти структурой данных, чем списки list.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
from collections import namedtuple

Data = namedtuple('Data', ['mathematics', 'chemistry', 'physics',
                           'genetics'], defaults=[5, 5, 5, 5])
d = {
    'Ivanov': Data(4, 5, 3, 5),
    'Petrov': Data(physics=4, genetics=3),
    'Sidorov': Data(),
}
```

5. Модуль argparse

В финале поговорим о модуле argparse. Мы упоминали его, когда речь шла о запуске скриптов с параметром через sys.argv. Поговорим о том чем же argparse лучше argv. Спойлер — всем. Модуль argparse по сути надстраивается над sys.argv. Он генерирует справку, определяет способ анализа аргументов командной строки, сообщает об ошибках и даёт подсказки. Чтобы разобраться во всём перечисленном по традиции рассмотрим простой пример.

```
import argparse

parser = argparse.ArgumentParser(description='My first argument
parser')
parser.add_argument('numbers', metavar='N', type=float,
nargs='*', help='press some numbers')
args = parser.parse_args()
print(f'В скрипт передано: {args}')
```



Внимание! Тут и далее до конца главы запускать файл будет из терминала ОС. Примерно так:

```
$ python main.py ...
```



где многоточие - передаваемые скрипту аргументы

Запустим скрипт с несколькими значениями:

```
$ python3 main.py 42 3.14 73
```

На выходе получаем объект `Namespace(numbers=[42.0, 3.14, 73.0])`. Как это работает?

1. Создаём объект парсер при помощи класса `ArgumentParser` с первоначальными настройками экземпляра.
2. Добавляем в полученный экземпляр аргументы для парсинга через метод `add_argument`. Количество аргументов может быть любым. И каждый может содержать свои настройки.
3. Выгружаем результаты, переданные при запуске скрипта в терминале и обработанные парсером в виде объекта `Namespace`. Для этого вызываем метод экземпляра `parse_args`.

Прежде чем разобрать каждый пункт более подробно запустим скрипт ещё пару раз: с ключом `--help` и с каким-нибудь текстом.

Ключ `--help` или `-h`

После создания экземпляра парсера и задания ему аргументов формируется справочный текст. `argparse` добавляет ключи `--help` (длинная версия) и `-h` (короткая версия) автоматически. Другие ключевые параметры мы можем создать сами, но о них чуть позже.

```
usage: main.py [-h] [N ...]

My first argument parser

positional arguments:
  N                press some numbers

options:
  -h, --help      show this help message and exit
```

Первая строка даёт общее представление о строке запуска. Ниже идёт текст, который мы указали в по ключу `description` при создании экземпляра. Далее получаем информацию о позиционных аргументах. В нашем случае это аргумент `N` (`metavar='N'`) и подсказки к нему (`help='press some numbers'`). В конце идёт необязательные параметры, в нашем случае - автоматически сгенерированный вызов помощи.

Запуск с неверными аргументами

При попытке передать в скрипт Hello world! получим:

```
usage: main.py [-h] [N ...]
main.py: error: argument N: invalid float value: 'Hello'
```

При создании аргумента мы указали, что хотим получать целые числа (`type=float`). Парсер автоматически создал валидатор и сообщил о несовпадении типов. Заметьте, что при передаче целых чисел ошибок не было, но они были преобразованы к вещественному типу.

Создаём парсер, `ArgumentParser`

При создании экземпляра из класса `ArgumentParser` можно 13 различных аргументов. Но большинство из них имеют оптимальные [настройки по умолчанию](#). Если что-то и стоит добавить, то дополнительное описание, которое выводится при вызове справки.

```
import argparse

parser = argparse.ArgumentParser(prog='average',
                                description='My first argument
parser',
                                epilog='Returns the arithmetic
mean')
...
```

- `prog` — заменяет название файла в первой строке справки на переданное имя,
- `description` — описание в начале справки
- `epilog` — описание в конце справки

Выгружаем результаты, parse_args

Метод parse_args может принимать на вход два аргумента:

- строку для анализа. По умолчанию это sys.argv
- объект для сохранения результатов. По умолчанию это класс Namespace. Класс наследуется от object, не имеет ничего лишнего, но добавляет удобный вывод ключей и значений, помещённых в него.

Изменять значения по умолчанию приходится крайне редко, почти никогда.

```
import argparse

parser = argparse.ArgumentParser(description='My first argument
parser')
parser.add_argument('numbers', metavar='N', type=float,
nargs='*', help='press some numbers')
args = parser.parse_args()
print(f'Получили экземпляр Namespace: {args = }')
print(f'У Namespace работает точечная нотация: {args.numbers =
}')
print(f'Объекты внутри могут быть любыми: {args.numbers[1] = }')
```

В этом случае код говорит сам за себя. Прочитайте три нижние строки. Уверен, что вопрос не должно остаться.

Добавляем аргументы, add_argument

А теперь самое интересное. Между созданием парсера и чтением результатов надо добавить желаемые аргументы.

Перед нами пример функции для решения квадратных уравнений. Параметры a, b, c собираем в терминале.

```
import argparse

def quadratic_equations(a, b, c):
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
    if d == 0:
        return -b / (2 * a)
    return None
```

```

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Solving
quadratic equations')
    parser.add_argument('param', metavar='a b c', type=float,
nargs=3,
                        help='enter a b c separated by a space')
    args = parser.parse_args()
    print(quadratic_equations(*args.param))

```

Вызов: `$ python3 main.py 2 -12 10`

Первая строка превращается в имя свойства. Если она начинается с одиночного или двойного дефиса, параметр считается необязательным. Далее:

- `metavar` — имя, которое выводится с справке
- `type` — тип, для преобразования аргумента. Тип помогает контролировать передачу нужных значений.
- `nargs` — указывает на количество значений, которые надо собрать из командной строки и собрать результат в список `list`. Целое число указывает количество. Кроме этого можно передать символ “?” — один аргумент, “*” — все имеющиеся аргументы, “+” — все имеющиеся аргументы, но не пустое значение.
- `help` - вывод подсказки об аргументе.

Если вызвать справку для нашего кода, увидим дублирование в первой строке
`usage: main.py [-h] a b c a b c a b c`

Необязательные аргументы и значения по умолчанию

Изменим наш парсер и добавим ещё несколько параметров к аргументам.

```

import argparse

def quadratic_equations(a, b, c):
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
    if d == 0:
        return -b / (2 * a)
    return None

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description='Solving

```

```
quadratic equations')
    parser.add_argument('-a', metavar='a', type=float,
help='enter a for ax^2+bx+c=0', default=1)
    parser.add_argument('-b', metavar='b', type=float,
help='enter b for ax^2+bx+c=0', default=0)
    parser.add_argument('-c', metavar='c', type=float,
help='enter c for ax^2+bx+c=0', default=0)
    args = parser.parse_args()
    print(quadratic_equations(args.a, args.b, args.c))
```

Вызов: \$ python main.py -a 2 -b -12

Теперь необходимо указывать ключи а, б и с при передаче значений. Но дополнительный параметр default позволяет отказаться от передачи. В этом случае значения будут взяты из параметра по умолчанию.

Параметр action для аргумента

И напоследок ещё один интересный параметр уже без квадратных уравнений. Речь пойдёт о параметре action.

```
import argparse

parser = argparse.ArgumentParser(description='Sample')
parser.add_argument('-x', action='store_const', const=42)
parser.add_argument('-y', action='store_true')
parser.add_argument('-z', action='append')
parser.add_argument('-i', action='append_const', const=int,
dest='types')
parser.add_argument('-f', action='append_const', const=float,
dest='types')
parser.add_argument('-s', action='append_const', const=str,
dest='types')
args = parser.parse_args()
print(args)
```

Вызов: \$ python3 main.py -x -y -z 42 -z 73 -i -f -s

Параметр action принимает одно из [определённых строковых значений](#) и срабатывает при наличии в строке вызова скрипта соответствующего параметра.

- store_const — передаёт в args ключ со значением из параметра const
- store_true или store_false — сохраняет в ключе истину или ложь
- append — ищет несколько появлений ключа и собирает все значения после него в список
- append_const — добавляет значение из ключа в список, если ключ вызван.

- параметр `dest` переопределяет имя ключа в `Namespace` на своё. В результате несколько разных ключей при вызове скрипта имеют одно имя при оценке результата.

Пожалуй это всё о основных способностях модуля `argparse`.

Вывод

Мы заканчиваем курс работой в командной строке. Тем, с чего начинали на первой лекции. Немного символично, как замыкание большого цикла знаний. Надеюсь вам было интересно и познавательно. Впереди ещё много нового и неизведанного. Но вы достаточно глубоко погрузились в Python, чтобы претендовать на ступеньку junior разработчика.

На этой лекции мы:

1. Узнали о составе стандартной библиотеки Python.
2. Разобрались в настройках логирования
3. Изучили работу с датой и временем
4. Узнали ещё пару полезных структур данных
5. Изучили способы парсинга аргументов при запуске скрипта с параметрами

Домашнее задание

Обратитесь к официальной документации по стандартной библиотеке Python. Выберите любой раздел по вашему желанию. Почитайте о модуле, попробуйте запустить примеры из документации.