Алгоритмизация и программирование. Перечень теоретических вопросов

***1. Парадигмы программирования их суть и сильные стороны. Типичные представители различных парадигм, применение различных парадигм в Python.***

Парадигма программирования - обобщение о том, как должна быть организована работа программы.

Императивное/структурное (директивное) – парадигма программирования, в основе которой лежит представление программы в виде иерархической структуры блоков. (Fortran, Pascal, C)

Объектно-ориентированное – представление программы в виде совокупности объектов. Объекты взаимодействуют между собой с помощью механизма передачи сообщений. (С++, Java, Python)

В основе ООП лежат такие понятия как **наследование**, **полиморфизм** и **инкапсуляция**.

Инкапсуляция предполагает, что малозначащие детали объекта скрыты. Объект, получая какую-либо команду, сам «знает» как ее обработать исходя из того, к какому классу он принадлежит.

Все объекты являются экземплярами классов, которые по отношению друг к другу могут выступать в роли родитель-потомок. Дочерние классы наследуют свойства родительского. В случае, когда 100% наследование не требуется, выручает так называемый полиморфизм, который предполагает переопределение методов родительского класса в дочерних классах.

Декларативное – это парадигма программирования, в которой задаётся спецификация решения задачи, то есть описывается, что представляет собой проблема и ожидаемый результат. (JavaScript)

Функциональное – парадигма программирования, в которой процесс вычисления трактуется как вычисление значений функций (R, Haskell, LISP)

Python поддерживает различные парадигмы программирования: императивное, объектно-ориентированное, и, частично, функциональное программирование.

***2. Специфика типизации в языках программирования (различные аспекты типизации). Реализация типизации в Python.***

Статическая – переменная, параметр подпрограммы, возвращаемое значение функции связывается с типом в момент объявления и тип не может быть изменен в процессе выполнения программы.

Динамическая – переменная связывается с типом в момент присваивания значения, тогда в разных участках программы одна и та же переменная может принимать разные типы.

Явная / неявная типизация. Явно-типизированные языки отличаются тем, что тип новых переменных / функций / их аргументов нужно задавать явно. Соответственно языки с неявной типизацией перекладывают эту задачу на компилятор / интерпретатор.

Сильная / слабая типизация (также иногда говорят строгая / нестрогая). Сильная типизация выделяется тем, что язык не позволяет смешивать в выражениях различные типы и не выполняет автоматические неявные преобразования, например нельзя вычесть из строки множество. Языки со слабой типизацией выполняют множество неявных преобразований автоматически, даже если может произойти потеря точности или преобразование неоднозначно.

В Python динамическая сильная неявная типизация

***3. Именование переменных и других объектов в Python (правила и соглашения). Числовые типы: литералы, объявление и операции.***

* Каждая переменная должна иметь уникальное имя, состоящее из латинских букв, цифр и знаков подчеркивания, причем имя переменной не может начинаться с цифры.
* При указании имени переменной важно учитывать регистр букв.
* Следует избегать указания символа подчеркивания в начале имени, т. к. идентификаторы с таким символом имеют специальное значение.
* В качестве имени переменной нельзя использовать ключевые слова.
* Имена переменных должны состоять из маленьких букв, а слова разделяться символами подчеркивания.

Для стандартизации записи кода в Python создан стандарт PEP8.

Для именования переменных или других объектов следует избегать символы l (маленькая латинская буква «эль»), O (заглавная латинская буква «о») или I (заглавная латинская буква «ай») как однобуквенные идентификаторы.

Модули и функции должны иметь короткие имена, состоящие из маленьких букв. Можно использовать символы подчеркивания, если это улучшает читабельность. То же самое относится и к именам пакетов, однако в именах пакетов не рекомендуется использовать символ подчёркивания.

Имена классов должны обычно следовать соглашению CapWords (каждое новое слово с заглавное буквы).

Для именования имен методов и переменных экземпляров классов используйте один символ подчёркивания перед именем для непубличных методов и атрибутов.

Чтобы избежать конфликтов имен с подклассами, используйте два ведущих подчеркивания.

***4. Присвоение по ссылке и по значению. Специфика создания объектов и присвоения в Python, особенности работы с объектами целочисленного типа.***

Для доступа к объекту предназначены переменные. **В языке Python все переменные являются ссылками на объекты.** В Python при инициализации в переменной сохраняется ссылка (адрес объекта в памяти компьютера) на объект. Благодаря этой ссылке можно в дальнейшем использовать объект из программы.

В Python для присвоения используется = (для сравнения значений на равенство используется ==).

* Первое присвоение значения переменной создает ее.
* Нет необходимости декларировать тип переменной (он определяется автоматически - по типу присваиваемого объекта).

***5. Разница между копированием и присвоением. Проблема утечки динамической памяти, сборка мусора. Копирование, присвоение и стратегия управления динамической памятью в Python.***

В Python изменяемые объекты нельзя скопировать, присвоив одну переменной другой, так как в этом случае копируется ссылка на объект, а не он сам. В итоге при изменении объекта через одну переменную, изменения видны через другую. Поэтому используются иные способы копирования. Если же требуется полная копия объекта, следует воспользоваться функцией deepcopy() модуля copy. Кроме этой функции там также есть функция copy(), выполняющая поверхностное копирование.

**Управление памятью и сборка мусора**

**Ручное управление памятью**

* Для создания объекта в динамической памяти программист явно вызывает команду выделения памяти. Эта команда возвращает указатель на выделенную область памяти, который сохраняется и используется для доступа к ней. До тех пор, пока созданный объект нужен для работы программы, программа обращается к нему через ранее сохранённый указатель.
* Когда надобность в объекте проходит, программист явно вызывает команду освобождения памяти, передавая ей указатель на удаляемый объект.
* В любом языке, допускающем создание объектов в динамической памяти, потенциально возможны две проблемы: висячие ссылки и утечки памяти.

**Сборка мусора**

* В системе со сборкой мусора (garbage collection, GC) обязанность освобождения памяти от объектов, которые больше не используются, возлагается на среду исполнения программы. Программист лишь создаёт динамические объекты и пользуется ими, он может не заботиться об удалении объектов. Для осуществления сборки мусора в состав среды исполнения включается специальный программный модуль, называемый "сборщиком мусора". Этот модуль периодически запускается, определяет, какие из созданных в динамической памяти объектов более не используются, и освобождает занимаемую ими память.
* Сборка мусора — технология, позволяющая, с одной стороны, упростить программирование, избавив программиста от необходимости вручную удалять объекты, созданные в динамической памяти, с другой — устранить ошибки, вызванные неправильным ручным управлением памятью (но не другими ресурсами).

***6. Булевский тип, сравнения и условные операторы в Python.***

Логический тип представлен двумя постоянными значениями False и True. Значения используются для представления истинности.

Для упрощения можно запомнить, что "объекты-контейнеры", не содержащие элементов (т.е. для которых len(obj) возвращает 0), приводятся к False.

Операции с логическим типом: not (операция отрицания), or (соответствует +), and (соответствует \*).

**Виды сравнений:**

a == b # проверка на равенство

a != b # проверка на неравенство

a < b # меньше

a <= b # меньше или равно

a > b # больше

a >= b # больше или равно

Автоматического приведения типов при сравнении (кроме случая числовых типов) НЕТ.

s1 is s2 # совпадают ли объекты?

**Условные операторы**

В языке Python для обозначения блочной структуры используются отступы (вместо фигурных скобок ({ и }) в языках с С-образным синтаксисом или блоков begin - end в Pascal).

В Python существует ключевое слово pass, которое представляет собой инструкцию, не делающую ровным счетом ничего, и которая может использоваться везде, где требуется блок кода (или когда мы хотим указать на наличие особого случая), но никаких действий выполнять не требуется.

Общий вид синтаксиса тернарного условного оператора:

true\_result **if** boolean\_expression1 **else** false\_result

***7. Циклы в Python, работа и устройство цикла for, типичное применение range и enumerate в цикле for.***

Синтаксис оператора цикла for:

**for** expression **in** iterable:

for\_suite

**else**:

else\_suite

* В качестве iterable может использоваться любой тип данных, допускающий выполнение итераций по нему, включая строки (где итерации выполняются по символам строки), списки, кортежи и другие типы коллекций языка Python.
* Блок else не обязательный
* Внутри цикла могут использоваться операторы break и continue
* В практике программирования на Python циклы for используются намного чаще чем while

**Функция range**

Функция range возвращает целочисленный итератор. Способы обращения к функции range:

* range(stop)

С одним аргументом (stop) итератор представляет последовательность целых чисел от 0 до (stop - 1).

* range(start, stop)

С двумя аргументами (start, stop) - последовательность целых чисел от start до (stop - 1).

* range(start, stop, step)

С тремя аргументами - последовательность целых чисел от start до (stop - 1) с шагом step.

Функция range позволяет удобно реализовать обход целочисленной последовательности с помощью оператора for.

for i in range(4):

print(i)

**Функция enumerate**

Функция enumerate обычно используется в циклах for ... in, чтобы получить последовательность кортежей (index, item), где значения индексов отсчитывается от 0 или от значения start;

Синтаксис функции enumerte:

* enumerate(i) генерируется последовательность кортежей (index, item), где значения индексов отсчитывается от 0
* enumerate (i, start) генерируется последовательность кортежей (index, item), где значения индексов отсчитывается от значения start

st = 'Hello world'

for ind, symb in enumerate(st):

print(f"Symbol '{symb}' has index {ind}")

***8. Строки в Python. Принципы работы и основные операторы и функции.***

Строки в Python являются неизменяемым типом данных содержащим массив символов.

Строку, введенную между утроенными апострофами или утроенными кавычками, можно разместить на нескольких строках, а также одновременно использовать кавычки и апострофы без необходимости их экранировать.

Если перед строкой разместить модификатор r (сокращение от raw), то специальные символы внутри строки выводятся как есть. Например, символ \n не будет преобразован в символ перевода строки. Иными словами, он будет считаться последовательностью двух символов: \ и n.

По сути строки являются неизменяемыми последовательностями (массивами) и все функциональные возможности, существующие у неизменяемых последовательностей, могут использоваться и у строк.

Получение среза с 3 параметрами: seq[start:end:step] (от символа с индексом start до символа с индексом end (не включая символ end !) с шагом step)

Проверка вхождения подстроки в строку с помощью оператора ‘in’

Можно использовать операцию конкатенации (склеивания) строк с помощью ‘+’

strip([<Символы>]) - удаляет пробельные или указанные символы в начале и конце строки.

lstrip( [<Символы>] ) - удаляет пробельные или указанные символы в начале строки

rstrip( [<Символы>] ) - удаляет пробельные или указанные символы в конце строки

split ( [<Разделитель> [, <Лимит> ]]) - разделяет строку на подстроки по указанному разделителю и добавляет их в список. Если первый параметр не указан или имеет значение None, то в качестве разделителя используется символ пробела. Если во втором параметре задано число, то в списке будет указанное количество подстрок. Если подстрок больше указанного количества, то список будет содержать ещё один элемент, в котором будет остаток строки.

splitlines( [True] ) - разделяет строку на подстроки по символу перевода строки (\n) и добавляет их в список. Символы новой строки включаются в результат, только если необязательный параметр имеет значение тrue. Если разделитель не найден в строке, то список будет содержать только один элемент.

join() - преобразует последовательность (в частности список) в строку. Элементы добавляются через указанный разделитель.

Формат метода: <Строка> = <Разделитель>.jоin(<Последовательность>)

Для изменения регистра символов предназначены следующие методы:

* upper() - заменяет все символы строки соответствующими прописными буквами;
* lower() - заменяет все символы строки соответствующими строчными буквами;
* swapcase() - заменяет все строчные символы соответствующими прописными буквами, а все прописные символы - строчными;
* capitalize() - делает первую букву прописной;
* title() - делает первую букву каждого слова прописной.
* функция chr(<Код символа>) - возвращает символ по указанному коду;
* функция ord(<Символ>) - возвращает код указанного символа.

**Поиск и замена в строке**

find() - ищет подстроку в строке. Возвращает номер позиции, с которой начинается вхождение подстроки в строку. Если подстрока в строку не входит, то возвращается значение -1. Метод зависит от регистра символов.

Формат метода: <Строка>.find(<Подстрока>[, <Начало>[, <Конец>]])

rfind() - ищет подстроку в строке. Возвращает позицию ПОСЛЕДНЕГО вхождения подстроки в строку. Если подстрока в строку не входит, то возвращается значение -1.

Параметры метода: <Строка>.rfind(<Подстрока>[, <Начало>[, <Конец>]])

index() - метод аналогичен методу find (), но если подстрока в строку не входит, то возбуждается исключение ValueError.

Параметры метода: <Строка>.indех(<Подстрока>[, <Начало>[, <Конец>]])

rindex() - метод аналогичен методу rfind(), но если подстрока в строку не входит, то возбуждается исключение Va1ueError.

Параметры метода: <Строка>.rindех(<Подстрока>[, <Начало>[, <Конец>]])

count() - возвращает число вхождений подстроки в строку. Если подстрока в строку не входит, то возвращается значение о. Метод зависит от регистра символов.

Параметры метода: <Строка>.соunt(<Подстрока>[, <Начало>[, <Конец>]])

startswith() - проверяет, начинается ли строка с указанной подстроки. Если начинается, то возвращается значение True, в противном случае - False. Метод зависит от регистра символов.

Параметры метода: < Cтpoкa \> .startswith(<Пoдcтpoкa>[, <Начало>[, <Конец>]])

endswith() - проверяет, заканчивается ли строка указанной подстрокой. Если заканчивается, то возвращается значение True, в противном случае - False. Метод зависит от регистра символов.

replace() - производит замену всех вхождений подстроки в строку на другую подстроку и возвращает результат в виде новой строки. Метод зависит от регистра символов.

Параметры метода: <Строка>.rер1асе(<Подстрока для замены>, <Новая подстрока>[, <Максимальное количество замен>])

**Проверка типа содержимого строки**

* isdigit() - возвращает тrue, если строка содержит только цифры, в противном случае - False;
* isdecimal() - возвращает True, если строка содержит только десятичные символы, в противном случае- False. Обратите внимание на то, что к десятичным символам относятся не только десятичные цифры в кодировке ASCII, но и надстрочные и подстрочные десятичные цифры в других языках.
* isnumeric() - возвращает True, если строка содержит только числовые символы, в противном случае- False. Обратите внимание на то, что к числовым символам относятся не только десятичные цифры в кодировке ASCII, но символы римских чисел, дробные числа и др.
* isalpha() - возвращает тrue, если строка содержит только буквы, в противном случае- False. Если строка пустая, то возвращается значение False.
* isspace() - возвращает тrue, если строка содержит только пробельные символы, в противном случае - False.
* isalnum() - возвращает True, если строка содержит только буквы и (или) цифры, в противном случае- False. Если строка пустая, то возвращается значение False.
* islower() - возвращает тrue, если строка содержит буквы, и они все в нижнем регистре, в противном случае- False. Помимо букв строка может иметь другие символы, например, цифры.
* isupper() - возвращает True, если строка содержит буквы, и они все в верхнем регистре, в противном случае- False. Помимо букв строка может иметь другие символы, например, цифры.
* istitle() - возвращает тrue, если строка содержит буквы, и первые буквы всех слов являются заглавными, в противном случае- False. Помимо букв строка может иметь другие символы, например, цифры.

***9. Списки в Python. Различные способы создания и копирования списков в Python. Обход списка и поиск элементов в списке.***

Списки - это нумерованные наборы объектов. Каждый элемент набора содержит лишь ссылку на объект. По этой причине списки могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Позиция элемента в наборе задается индексом. Нумерация элементов, как и в строках, начинается с 0, а не с 1.

* Создание пустого списка с помощью конструктора:
  + lst\_1 = list()
* Создание списка с уже заданными значениями:
  + lst\_2 = [2, True, 'my string', '7']
* Создание пустого списка и добавление элементов в конец списка:
  + lst\_3 = []
  + lst\_3.append(1)
  + lst\_3.append(False)
  + lst\_3.append('my\_stirng')
* list() может создать список из любого итерируемого объекта:
  + lst\_4 = list('Hello world')
* Список может создаваться на основе функций или объектов, которые возвращают последовательность значений:
  + lst\_5 = list(range(5, 10)

**Вложенные списки:**

Вложенные списки - это полноценные независимые объекты класса list, поэтому:

* вложенные списки могут менять размер независимо от других списков
  + это может приводить к превращению вложенных списков к "не прямоугольному" виду
* объекты, хранящиеся во вложенных списках, могут иметь произвольный тип

Создание вложенных списков: lst\_6 = [[1, 0, 0], [0, 1, 0], [0, 0, 1]]

**Копирование списков**

При создании списка в переменной сохраняется ссылка на объект, а не сам объект. Это обязательно следует учитывать при групповом присваивании. Групповое присваивание безопасно использовать для чисел и строк, но для списков это может приводить к ошибкам.

*Создание поверхностной копии списка:*

Создание списка 🡪 lst\_from = [1, 2, 3, 4, 5]

Создание нового списка на основе объекта, который возвращает последовательность значений 🡪 lst\_to = list(lst\_from)

Поверхностное копирование вложенных списков не приводит к копированию вложенных списков.

**Создание глубокой копии списков**

Подключаем модуль copy 🡪 import copy

Делаем полную копию списка 🡪 lst\_to2\_deep = copy.deepcopy(lst\_from2)

**Обход списка**

for el in lst6:

print(el, end=' ')

Можно использовать функцию *enumerate*, что позволяет удобно обходить список и оперировать с текущим индексом элемента:

for i, el in enumerate(lst6):

print(f'индекс: {i}, значение: {el}')

**Поиск элемента в списке**

Метод index() позволяет узнать индекс элемента с определенным значением.

Формат метода: indех(<Значение> [,<Начало> [,<Конец>]]). Метод index() возвращает индекс элемента, имеющего указанное значение.

* Если значение не входит в список, то возбуждается исключение ValueError.
* Если второй и третий параметры не указаны, то поиск будет производиться с начала списка.

***10. Списки в Python. Обращение к элементам списка и создание срезов. Стандартные агрегирующие функции, работающие со списками.***

len() - встроенная функция, которая определяет длину списка. Ее не нужно импортировать, она доступна всегда.

Как и со строковыми переменными, можно обращаться к отдельным элементам:

lst[ <номер элемента> ]

Последний элемент списка имеет индекс на 1 меньший чем длина строки, а нумерация элементов начинается с 0.

**Извлечение среза** (Срезы работаю, как и для строк)

Списки поддерживают **операцию извлечения среза**, которая возвращает указанный фрагмент списка. Формат операции: [<Начало>:<Конец>:<Шаг>]

Все параметры в операции среза являются необязательными. В частности, если:

* НЕ указан параметр <Начало>, то используется значение 0
* НЕ указан параметр <Конец>, то возвращается фрагмент до конца списка. Следует также заметить, что элемент с индексом, указанном в этом параметре, не входит в возвращаемый фрагмент.
* НЕ указан параметр <Шаг>, то используется значение 1. При этом второе двоеточие в срезе не используется.

В качестве значения параметров можно указать отрицательные значения.

**Агрегирующие функции**

sum(iter, start=0) - Сумма членов последовательности.

max(iter, [args ...] \* [, key]) - Максимальный элемент последовательности.

min(iter, [args ...] \* [, key]) - Минимальный элемент последовательности.

***11. Списки в Python. Ключевые операции, проводящие к изменению списка и порождающие измененные списки.***

**Изменение списка**

.append(<Объект>) - добавляет один объект в конец списка. Метод изменяет текущий сnисок и ничего не возвращает.

Изменение размера списков Python за счет добавления элементов в конец (или удаления последних элементов списка) является наиболее эффективной (с точки зрения затраты вычислительных ресурсов) операцией изменения списков.

.extend(<Последовательность>) - добавляет элементы последовательности в конец списка. Метод изменяет текущий список и ничего не возвращает.

.insert (<Индекс>, <Объект>) - добавляет один объект в указанную позицию. Остальные элементы смещаются. Метод изменяет текущий список и ничего не возвращает. Метод insert() позволяет добавить только один объект. Чтобы добавить несколько объектов, можно воспользоваться операцией присваивания значения срезу.

С помощью среза можно изменить фрагмент списка.

Вставка нескольких элементов при помощи среза (без потери существующих элементов):

In: lst4 = [3, 7, 13, 1]

lst4[1:1] = ['a', 'b', 'c']

Out: [3, 'a', 'b', 'c', 7, 13, 1]

Можно вставлять несколько элементов при помощи среза, замещая уже существующие эл-ты.

Если срезу присвоить пустой список, то элементы, попавшие в срез, будут удалены.

.рор([<Индекс>]) - удаляет элемент, расположенный по указанному индексу, и возвращает его.

* Если индекс не указан, то удаляет и возвращает последний элемент списка.
* Если элемента с указанным индексом нет или список пустой, возбуждается исключение IndexError.

.remove (<Значение>) - удаляет первый элемент, содержащий **указанное значение**. Если элемент не найден, возбуждается исключение ValueError. Метод изменяет текущий список и ничего не возвращает.

**Инструкция del** может удалять из списка как единичные элементы, так и элементы, получаемые при помощи среза.

del lst4[:2] или del lst4[2]

**Проверка на вхождение элемента в список**

Оператор in осуществляет проверку на вхождение элемента в список. Если элемент входит в список, то возвращается значение True, в противном случае - False. Оператор in не дает никакой информации о местонахождении элемента внутри списка.

**Сортировка**

Отсортировать список позволяет метод sort(). Метод имеет формат sort([key=None] [, reverse=fa1se]). Все параметры являются необязательными. Метод изменяет текущий список и ничего не возвращает.

* использование аргумента key будет рассмотрено в других лекциях

В некоторых случаях необходимо получить отсортированный список, а текущий список оставить без изменений. Для этого следует воспользоваться функцией sorted().

Функция имеет формат: sоrtеd(<Последовательность>[, key=None] [, reverse=False])

* В первом параметре указывается список, который необходимо отсортировать.
* Остальные параметры эквивалентны параметрам метода sort().

**Инвертирование списка**

Операция переворачивает список изменяя исходный объект:

lst8.reverse()

Если необходимо изменить порядок следования и получить новый список, то следует воспользоваться функцией reversed(<Последовательность>).

Функция возвращает итератор, который можно преобразовать в список с помощью функции list().

***12. Словари в Python. Основные способы создания, получения и изменения значений. Обход словарей.***

Словари - это наборы объектов, доступ к которым осуществляется не по индексу, а пo ключу. В качестве ключа можно указать неизменяемый объект, например число, строку или кортеж. Элементы словаря могут содержать объекты произвольного типа данных и иметь неограниченную степень вложенности. Элементы в словарях располагаются в произвольном порядке. Чтобы получить элемент, необходимо указать ключ, который использовался при сохранении значения.

**Создание словаря**

Создание словаря с nомощью конструктора dict(). Допустимые форматы вызова конструктора:

* diсt(<Ключ1>=<Значение1>[, ... , <КлючN>=<ЗначениеN>])
* diсt(<Словарь>)
* diсt(<Список кортежей с двумя элементами (Ключ, Значение)>)
* diсt(<Список списков с двумя элементами [Ключ, Значение]>)

Создание словаря при помощи указания всех элементов словаря внутри фигурных скобок. Это наиболее часто используемый способ создания словаря. Между ключом и значением указывается двоеточие, а пары "ключ/значение" перечисляются через запятую.

d0 = {'a': 1, 'b': 2}

С помощью функции zip() можно создать словарь из двух списков:

k = ['a', 'b', 'c']

v = [1, 2, 3]

dict(zip(k, v))

OUT: {'a': 1, 'b': 2, 'c': 3}

Создание словаря при помощи поэлементного заполнения словаря.

d3 = {} # создание пустого словаря

d3['a'] = 1 # добавление элемента в словарь

d3['b'] = 2

Ключами словаря могут быть только неизменяемые объекты (кортежи, строки, числа и т.д.)

Метод dict.fromkeys(<Последовательность> [, <Значение>]) создает новый словарь, ключами которого будут элементы последовательности. Если второй параметр не указан, то значением элементов словаря будет значение None.

Метод get(<Ключ> [, <Значение no умолчанию>]) позволяет избежать вывода сообщения об ошибке при отсутствии указанного ключа:

* если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу
* если ключ отсутствует, то возвращается значение None или значение, указанное во втором nараметре.

Кроме того, можно воспользоваться методом setdefault(<Ключ>[, <Значение по умолчанию>]):

* если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу
* если ключ отсутствует, то вставляет новый элемент, со значением, указанным во втором nараметре
  + если второй параметр не указан, значением нового элемента будет None.

Так как словари относятся к изменяемым типам данных, мы можем изменить элемент по ключу. Если элемент с указанным ключом отсутствует в словаре, то он будет добавлен в словарь.

d11['a'] = 11 # изменение значения по ключу

d11['с'] = 13 # добавление значения

Получить количество ключей в словаре позволяет функция len():

Удалить элемент из словаря можно с помощью оператора del:

**Перебор элементов словаря**

for k in d13:

print(f'key: {k}, value: { d13[k]}')

Получение cписка ключей 🡪 list(d13.keys())

Получение cписка значений 🡪 list(d13.values())

Обход пар ключ-значение в цикле for:

for k, v in d13.items():

print(f'key: {k}, value: {v}')

Методы dict.items(), dict.keys() и dict.values() возвращают **представления словарей**.

**Представление словаря** - это итерируемый объект, доступный только для чтения и хранящий элементы, ключи или значения словаря в зависимости от того, какое представление было запрошено.

Между представлениями и обычными итерируемыми объектами есть два различия:

* если словарь, для которого было получено представление, изменяется, то представление будет отражать эти изменения.
* представления ключей и элементов поддерживают некоторые операции, свойственные множествам. Допустим, у нас имеются представление словаря v и множество или представление словаря х; для этой пары поддерживаются следующие операции:
  + v & х # Пересечение
  + v | х # Объединение
  + v - х # Разность
  + v ^ х # Строгая дизъюнкция

**Методы**

.рор(<Ключ> [, <Значение по умолчанию>]) - удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, то возвращается значение из второго параметра. Если ключ отсутствует и второй параметр не указан, то возбуждается исключение KeyError.

.popitem() - удаляет произвольный элемент и возвращает кортеж из ключа и значения. Если словарь пустой, возбуждается исключение KeyError.

.clear() - удаляет все элементы словаря. Метод ничего не возвращает в качестве значения.

. update() - добавляет элементы в словарь. Метод изменяет текущий словарь и ничего не возвращает.

Форматы метода:

* uрdаtе(<Ключ1>=<Значение1>[, ... , <КлючN>=<ЗначениеN>])
* uрdаtе(<Словарь>)
* update(<Cпиcoк кортежей с двумя элементами>)
* update(<Cпиcoк списков с двумя элементами>)

Если элемент с указанным ключом уже присутствует в словаре, то его значение будет перезаписано.

***13. Преобразование между словарями и списками в Python. Операции с представлениями словарей.***

Создание списка на основе списка списков вида: (ключ, значение):

dict([['a', 1], ['b', 1]])

С помощью функции zip() можно создать словарь из двух списков:

k = ['a', 'b', 'c']

v = [1, 2, 3]

dict(zip(k, v))

**Операции с представлениями словарей:**

Получение cписка ключей 🡪 list(some\_dict.keys())

for k in d13.keys():

print(f'key: {k}, value: { d13[k]}')

Получение cписка значений 🡪 list(some\_dict.values())

for v in d13.values():

print(f'value: {v}')

Получение cписка кортежей ключ-значение 🡪 list(some\_dict.items())

Обход пар ключ-значение в цикле for:

for k, v in d13.items():

print(f'key: {k}, value: {v}')

***14. Операции со словарями, учитывающие возможное отсутствие ключа. Операции многоэлементного изменения словарей. Операции поэлементного извлечения из словаря и их использование.***

Кроме того, можно воспользоваться методом setdefault(<Ключ>[, <Значение по умолчанию>]):

* если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу
* если ключ отсутствует, то вставляет новый элемент, со значением, указанным во втором nараметре
  + если второй параметр не указан, значением нового элемента будет None.

Метод get(<Ключ> [, <Значение no умолчанию>]) позволяет избежать вывода сообщения об ошибке при отсутствии указанного ключа:

* если ключ присутствует в словаре, то метод возвращает значение, соответствующее этому ключу
* если ключ отсутствует, то возвращается значение None или значение, указанное во втором nараметре.

Обход упорядоченных ключей:

for k in sorted(d13.keys()):

print('key: {}, value: {}'.format(k, d13[k]))

Обход значений словаря:

for v in d13.values():

print('value: {}'.format(v))

***15. Множества в Python. Основные способы создания, получения и изменения значений. Обход множеств.***

Множество - это неупорядоченная коллекция уникальных элементов, с которой можно сравнивать другие элементы, чтобы определить, принадлежат ли они этому множеству. Множество может содержать только элементы неизменяемых типов, например числа, строки, кортежи. Объявить множество можно с помощью конструктора set().

Преобразуем список в множество 🡪 s2 = set ([1, 2, 3, 4, 5])

OUT: {1, 2, 3, 4, 5}

Преобразуем список в множество 🡪 s3 = set ([1, 2, 3, 1, 2, 3])

OUT: {1, 2, 3}

Использование итерируемого объекта вместо списка 🡪 s21 = set(range(5))

OUT: {0, 1, 2, 3, 4}

В Python 3 можно также создать множество, указав элементы внутри фигурных скобок. Обратите внимание на то, что при указании пустых фигурных скобок будет создан словарь, а не множество. Чтобы создать пустое множество, следует использовать функцию set().

Обход элементов множества в цикле for:

for v in s2:

print(v, end=' ')

OUT: 1 2 3 4 5

.add(<Элемент>) - добавляет <Элемент> во множество

.remove(<Элемент>) - удаляет <Элемент> из множества. Если элемент не найден, то возбуждается исключение KeyError

.disсаrd(<Элемент>)- удаляет <Элемент> из множества, если он присутствует

.рор() - удаляет произвольный элемент из множества и возвращает его. Если элементов нет, то возбуждается исключение KeyError

.clear() - удаляет все элементы из множества

in - проверка наличия элемента во множестве:

s3 = {1, 2, 3, 4}

2 in s3

OUT: True

== - проверка на равенство (совпадение значений множеств):

set([1, 2, 3]) == set([1, 2, 3])

OUT: True

* а <= b и a.issubset(b) - проверяют, входят ли все элементы множества а во множество b
* а < b - проверяет, входят ли все элементы множества а во множество b. Причем множество а не должно быть равно множеству b
* а >= b и а.issuperset(b) - проверяют, входят ли все элементы множества b во множество а
* а > b - проверяет, входят ли все элементы множества b во множество а. Причем множество а не должно быть равно множеству b
* а.isdisjoint(b) - возвращает True, если результатом пересечения множеств а и b является пустое множество (это означает, что множества не имеют одинаковых элементов)

.сору() - создает копию множества. Обратите внимание на то, что оператор = присваивает лишь ссылку на тот же объект, а не копирует его.

***16. Выполнение основных операций с парой множеств в Python.***

s3 | s4 и s3.union(s4) - объединение множеств:

s5 |= s4 и s5.update(s4) - добавляют элементы одного множества в другое

s4 & s3 и s4.intersection(s3) - пересечение множеств

а &= b и а.intersection\_update(b) - во множестве а останутся элементы, которые существуют и во множестве а, и во множестве b

s4 - s3 и s4.difference(s3) - разница множеств

s6 -= s3 и s6.difference\_update(s3) - удаляем элементы из множества а, которые присутствуют во множестве b

^ и symmetric\_difference() - возвращают элементы обоих множеств, присутствующие только в одном из множеств-аргументов.

a ^= b и a.symmetric\_difference\_update(b) - во множестве а будут все элементы обоих множеств, исключая одинаковые элементы.

***17. Кортежи в Python. Отличия кортежей от списков. Распаковка и частичная распаковка кортежей.***

Кортежи, так же как и списки, являются упорядоченными последовательностями элементов. Кортежи во многом аналогичны спискам, но имеют одно очень важное отличие - изменить кортеж нельзя. Можно сказать, что кортеж - это список, доступный "только для чтения".

Создание кортежа при помощи функции tuple([<Последовательность>]). Функция позволяет преобразовать любую последовательность в кортеж. Если параметр не указан, то создается пустой кортеж.

Создание пустого кортежа: tuple()

Создание кортежа из списка: tuple([1, 2, 3, 4])

Создание кортежа с помощью range(): tuple(range(10))

Создание списка из кортежа: lst9 = list((1, 2, 3, 4))

**Распаковка и частичная распаковка кортежей.**

Когда справа от оператора присваивания указывается последовательность (в данном случае - это кортежи), а слева указан кортеж, мы говорим, что последовательность справа **распаковывается**.

a, b, c = (10, 11, 12)

print(f'a = {a}, b = {b}, c = {c}')

OUT: a = 10, b = 11, c = 12

**Существует возможность частичной распаковки:**

Символ звездочка может находиться в любом месте, но не более одного раза:

a, \*b, c = range (10, 16)

print(f'a = {a}, b = {b}, c = {c}')

OUT: a = 10, b = [11, 12, 13, 14], c = 15

a, \*b, c = range (10, 12)

print(f'a = {a}, b = {b}, c = {c}')

OUT: a = 10, b = [], c = 11

***18. Выражения генераторы и генераторы списков в Python. Использование условий в генераторах.***

*Выражения генераторы*

lst\_val = [1, 2, 7, 11, 8, 2]

cor\_gen = (el \* 2 for el in lst\_val)

sum(cor\_gen)

OUT: 62

Передаем генератор напрямую: sum((el \* 2 for el in lst\_val))

*Генераторы списков*

Создание списка при помощи генератора:

lst\_val = [1, 2, 7, 11, 8, 2]

lst\_gen = [el \* 2 for el in lst\_val]

OUT: [2, 4, 14, 22, 16, 4]

Создание списка при помощи генератора и условия:

lst\_gen2 = [el \* 2 for el in lst\_val if el % 2 == 0]

OUT: [4, 16, 4]

***19. Генераторы множеств и словарей в Python. Использование условий в генераторах.***

**Генераторы множеств**

Помимо генераторов списков язык Python поддерживает генераторы множеств. Синтаксис генераторов множеств похож на синтаксис генераторов списков, только вместо квадратных скобок используются фигурные скобки.

sg1 = {х\*\*2 for х in [1, 2, 1, 2, 1, 2, 3]}

OUT: {1, 4, 9}

sg2 = {x for x in [1, 2, 1, 2, 1, 2, 3] if x % 2 == 0}

OUT: {2}

**Генераторы словарей**

Помимо генераторов списков язык Python поддерживает генераторы словарей. Синтаксис генераторов словарей похож на синтаксис генераторов списков, но имеет два отличия:

* выражение заключается в фигурные скобки, а не в квадратные;
* внутри выражения перед циклом for указываются два значения через двоеточие, а не одно:
  + значение, расположенное слева от двоеточия: ключ
  + значение, расположенное справа от двоеточия: значение элемента.

dg1 = {e: e \*\* 2 for e in range(10)}

dg1

OUT: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}

d18 = {'a': 1, 'b': 2}

dg2 = {k: 2 \* v for k, v in d18.items() if v % 2 == 0}

dg2

OUT: {'b': 4}

***20. Функции стандартной библиотеки для работы с контейнерами.***

ПРОДОЛЖИТЬ С ЛЕКЦИИ №4

Проверить существование ключа можно с помощью оператора in:

* если ключ найден, то возвращается значение True
* в противном случае - False.

Оператор is возвращает истину, если оба операнда указывают на один объект.

Функция isinstance() специально создана для проверки принадлежности данных определенному классу (типу данных):

a = 10

isinstance(a,int)

OUT: True

***21. Объявление и вызов функции в Python. Параметры функции со значением по умолчанию и комментирование функции. Получение информации о функции. Способы передачи параметров при вызове функции.***

Функция описывается с помощью ключевого слова def по следующей схеме:

def <Имя функции> ([<Параметры>]):

[""" Строка документирования """]

<Тело функции>

[return <Значение>]

* **Имя функции** должно быть корректным уникальным идентификатором
  + состоящим из латинских букв, цифр и знаков подчеркивания (причем имя функции не может начинаться с цифры).
  + В качестве имени нельзя использовать ключевые слова, кроме того, следует избегать совпадений с названиями встроенных идентификаторов.
  + Регистр символов в названии функции имеет значение.
* После имени функции в круглых скобках можно указать один или несколько **параметров** через запятую.
  + Если функция не принимает параметры, то просто указываются круглые скобки.
  + После круглых скобок ставится двоеточие.
* После двоеточия может следовать необязательная **строка документирования** функции (распространено использование многострочной строки, заключенной в тройные кавычки)
* **Тело функции** является составной конструкцией. Как и в любой составной конструкции, инструкции внутри функции выделяются одинаковым количеством пробелов слева.

Пример вызова функции:

def func():

print ("Текст до инструкции return")

return "Возвращаемое значение"

print ("Эта инструкция никогда не будет выполнена")

print(func())

OUT: Текст до инструкции return

Возвращаемое значение

Необязательная инструкция return позволяет вернуть значение из функции. После исполнения этой инструкции выполнение функции будет завершено.

Строку документирования можно вызвать с помощью функции help() или через метод \_\_doc\_\_.

def echo(m):

""" Пример функции с параметром """

print(m)

echo.\_\_doc\_\_

OUT: ' Пример функции с параметром '

help(echo)

OUT: Help on function echo in module \_\_main\_\_:

echo(m)

Пример функции с параметром

Информацию о функции также можно получить через атрибут ‘?’ в Jupyter

?echo

**Signature:** echo**(**m**)**

**Docstring:** Пример функции с параметром

**File:** c:\users\asus\desktop\вуз\python\лекции\lec04\_functions\<ipython-input-1-70627dcbb54d>

**Type:** function

Функции в Python можно передавать в качестве аргументов других функций:

def func(fp, а, b):

""" Через переменную fp будет доступна ссылка на функцию summa() """

return fp(а, b) # Вызываем функцию summa()

func(summa, 10, 20)

OUT: 30

Чтобы сделать некоторые параметры необязательными, следует в определении функции присвоить этому параметру начальное значение.

def summa(x, y=2):

return x + y

summa(5)

OUT: 7

summa(5, 4)

OUT: 9

**Глобальные переменные** - это переменные, объявленные в программе вне функции. В Pythoп глобальные персменные видны в любой части модуля, включая функции.

**Локальные переменные** - это переменные, которым внутри функции присваивается значение. Если имя локальной переменной совпадает с именем глобальной переменной, то все операции внутри функции осуществляются с локальной переменной, а значение глобальной переменной не изменяется. Локальные персменные видны только внутри тела функции.

Для того чтобыI значение глобальной переменной можно было изменить внутри функции, необходимо объявить переменную глобальной с помощью ключевого слова *global.*

Получить все идентификаторы и их значения nозволяют следующие функции:

* globals() - возвращает словарь с глобальными идентификаторами;
* locals() - возвращает словарь с локальными идентификаторами;
* vars( [<Объект>] ) - если вызывается без параметра внутри функции, то возвращает словарь с локальными идентификаторами. Если вызывается без параметра вне функции, то возвращает словарь с глобальными идентификаторами. При указании объекта в качестве параметра возвращает идентификаторы этого объекта (эквивалентно вызову <Объект>.\_\_dict\_\_ )

***22. Передача переменного количества параметров (именованных и не именованных) в функции Python. Вызов функции с позиционными параметрами, находящимися в списке, и именованными параметрами, находящимися в словаре.***

def multiply(\*args):

if len(args) == 0:

return

else:

rez = 1

for i in args:

rez \*= i

return rez

Данная функция возвращает произведение полученных параметров. \*args гласит, что может быть использовано неограниченное количество неименованных параметров

print(multiply(1, 2, 34, 5, 8))

OUT: 2720

print(multiply())

OUT: None

def str\_multiply(\*\*kwargs):

if len(kwargs) == 0:

return

else:

string = ''

for k, v in kwargs.items():

string += k\*v

return string

Данная функция принимает неограниченное количество именованных параметров при помощи \*\*kwargs

str\_multiply(a=5, b=10)

OUT: 'aaaaabbbbbbbbbb'

def multi\_summa(x=1, y=1, z=1):

return x + 10\* y + 100\* z

Если значения параметров, которые планируется передать в функцию, содержатся в кортеже или списке, то перед объектом следует указать символ \* .

Распаковка:

t1 = (5, 10 ,15)

multi\_summa(\*t1)

OUT: 1605

!!! Количество передаваемых параметров должно равняться количеству объявленных в функции параметров:

t3 = (5, 10 ,15, 20)

multi\_summa(\*t3)

OUT: TypeError

Если значения параметров содержатся в словаре, то распаковать словарь можно, указав перед ним две звездочки \*\* :

d1 = {'x': 11, 'y': 12, 'z': 13}

multi\_summa(\*\*d1)

OUT: 1431

***23. Анонимные функции в Python их возможности и ограничения. Типичные сценарии использования анонимных функций.***

Помимо обычных функций язык Python позволяет использовать анонимные функции, которые называются **лямбда-функциями**. Анонимная функция описывается с помощью ключевого слова lambda по следующей схеме:

lamda [<Параметрl>[, ... , <ПараметрN>]]: <Возвращаемое значение>

После ключевого слова lambda можно указать передаваемые параметры. В качестве параметра <Возвращаемое значение> указывается выражение, результат выполнения которого будет возвращен функцией.

Как видно из схемы, у лямбда-функций нет имени. По этой причине их и называют анонимными функциями.

В качестве значения **лямбда-функция возвращает ссылку на объект-функцию**, которую можно сохранить в переменной или передать в качестве параметра в другую функцию. Вызвать лямбда-функцию можно, как и обычную, с помощью круглых скобок, внутри которых расположены передаваемые параметры.

Функция без параметров:

fl = lambda: 10 + 20

print(fl())

OUT: 30

Функция с двумя параметрами:

f2 = lambda х, y: х + y

print(f2(5, 10))

OUT: 15

Функция с тремя параметрами:

f3 = lambda х, y, z: х + y + z

print(f3(5, 10, 30))

OUT: 45

Данную функцию можно использовать в рамках каких-либо **конвейерных вычислений** (например внутри filter(), map() и reduce()) либо самостоятельно, в тех местах, где требуется произвести какие вычисление, которые удобно “завернуть” в функцию.

***24. Синтаксис и семантика обработки исключительных ситуаций в Python.***

Исключения - это извещения интерпретатора, возбуждаемые в случае возникновения ошибки в программном коде или при наступлении какого-либо события. Если в коде не предусмотрена обработка исключения, то программа прерывается и выводится сообщение об ошибке.

Существуют три типа ошибок в программе:

* синтаксические - это ошибки в имени оператора или функции, отсутствие закрывающей или открывающей кавычек и т. д., т. е. ошибки в синтаксисе языка. Как правило, интерпретатор предупредит о наличии ошибки, а программа не будет выполняться совсем.

Пример синтаксической ошибки: print("Heт завершающей кавычки!)

* логические - это ошибки в логике работы программы, которые можно выявить только по результатам работы скрипта. Как правило, интерпретатор не предупреждает о наличии ошибки. А программа будет выполняться, т. к. не содержит синтаксических ошибок. Такие ошибки достаточно трудно выявить и исправить;
* ошибки времени выполнения - это ошибки, которые возникают во время работы скрипта. Причиной являются события, не предусмотренные программистом. Классическим примером служит деление на ноль

**Инструкция try ... except ... else ... finally**

Для обработки исключений предназначена инструкция try. Формат инструкции:

try:

<Блок, в котором nерехватьшаются исключения>

[ except [ <Исключение1> [ as <Объект исключения>] ] :

<Блок, вьmолняемьм nри возникновении исключения>

[ ...

except [<ИсключениеN>[ as <Объект исключения>]]:

<Блок, вьmолняемьм nри возникновении исключения>]]

[e1se:

<Блок, вьmолняемьм, если исключение не возникло>]

[finally:

<Блок, вьmолняемьм в любом случае>]

Инструкции, в которых перехватываются исключения, должны быть расположены внутри блока try. В блоке except в параметре <Исключение1> указывается класс обрабатываемого исключения. Например, обработать исключение, возникающее при делении на ноль, можно так:

x = 0

try: # Перехватаем исключения

х = 1/0 # Ошибка: деление на 0

except ZeroDivisionError: # Указываем класс исключения

print("Обработали деление на 0")

Если в блоке try возникло исключение, то управление передается блоку except. В случае, если исключение не соответствует указанному классу, управление передается следующему блоку except. Если ни один блок except не соответствует исключению, то исключение "всплывает" к обработчику более высокого уровня. Если исключение нигде не обрабатывается в программе, то управление передается обработчику по умолчанию, который останавливает выполнение программы и выводит стандартную информацию об ошибке

В инструкции except можно указать сразу несколько исключений, перечислив их через запятую внутри круглых скобок:

try: # Обрабатьшаем исключения

х = 1/0 # Ошибка: деление на 0

except (NameError, IndexError, ZeroDivisionError):

х = 0

Если в инструкции except не указан класс исключения, то такой блок перехватывает все исключения. (на практике следует избегать, т. к. можно перехватить исключение, которое является лишь сигналом системе, а не ошибкой).

Если в обработчике присутствует блок else, то инструкции внутри этого блока будут выполнены только при отсутствии ошибок. При необходимости выполнить какие-либо завершающие действия вне зависимости от того, возникло исключение или нет, следует воспользоваться блоком finally.

***25. Создание пользовательских исключений и инструкция assert.***

**Пользовательские исключения**

Инструкция raise возбуждает указанное исключение. Она имеет несколько форматов:

* raise <Экземnляр класса>
* raise <Название класса>
* raise <Экземnляр или название класса> from <Объект исключения>
* raise

В первом формате инструкции raise указывается экземпляр класса возбуждаемого исключения. При создании экземпляра можно передать данные конструктору класса. Эти данные будут доступны через второй параметр в инструкции except.

try:

raise ValueError("Onиcaниe исключения")

except ValueError as msg:

print(msg)

OUT: Описание исключения

try:

raise ValueError

except ValueError:

print('Сообщение об ошибке')

OUT: Описание исключения

**Инструкция assert**

Инструкция assert возбуждает исключение AssertionError, если логическое выражение возвращает значение False. Инструкция имеет следующий формат:

assert <Логическое выражение> [, <Сообщение>]

Инструкция assert эквивалентна следующему коду:

if \_\_debug\_\_:

if not <Логическое выражение>:

raise AssertionError(<Сообщение>)

Если при запуске программы используется флаг -о, то переменная \_\_debug\_\_ будет иметь ложное значение. Таким образом можно удалить все инструкции assert из байт-кода.

def factorial(n):

"""Возвращает Факториал числа n. Аргумент n - не отрицательное целое число."""

assert n >= 0, 'Аргумент n должен быть больше 0!'

assert n % 1 == 0, 'Аргумент n должен быть целым!'

f = 1

for i in range(2, n+1):

f \*= i

return f

factorial(-1)

OUT: **AssertionError**: Аргумент n должен быть больше 0!

factorial(5.5)

OUT: **AssertionError**: Аргумент n должен быть целым!

factorial(3)

OUT: 6

***26. Базовые операции для работы с файлами в Python.***

Файл — именованная область данных на носителе информации. Работа с файлами реализуется средствами операционных систем. Операционная система предоставляет приложениям набор функций и структур (API) для работы с файлами. На уровне API уже не существенно, существует ли файл как объект файловой системы или является, например, устройством ввода-вывода.

Можно выделить два типа операций с файлом — связанные с его открытием и выполняющиеся без его открытия.

* Операции первого типа обычно служат для чтения и записи информации или подготовки к чтению или записи.
* Операции второго типа выполняются с файлом как с «объектом» файловой системы, в котором файл является наименьшим элементом структурирования.

**Базовые операции для работы с файлом:**

За открытие файла отвечает функция open.

Синтаксис - open(filepath, режим, encoding='utf-8'[,доп.режимы]).

После того, как вы открыли файл и сделали все необходимы операции, вам необходимо закрыть его функции close()

Необязательный параметр mode в функции open() может принимать следующие значения:

* r - только чтение (значение по умолчанию). После открытия файла указатель устанавливается на начало файла. Если файл не существует, то возбуждается исключение IOError;
* r+ - чтение и запись. После открытия файла указатель устанавливается на начало файла. Если файл не существует, то возбуждается исключение IOError;
* w - запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
* w+ - чтение и запись. Если файл не существует, то он будет создан. Если файл существует, то он будет перезаписан. После открытия файла указатель устанавливается на начало файла;
* а - запись. Если файл не существует, то он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется;
* а+ - чтение и запись. Если файл не существует, то он будет создан. Запись осуществляется в конец файла. Содержимое файла не удаляется.

Кроме того, после режима может следовать модификатор:

* b - файл будет открыт в бинарном режиме. Файловые методы принимают и возвращают объекты типа bytes;
* t - файл будет открыт в текстовом режиме (значение по умолчанию в Windows). Файловые методы принимают и возвращают объекты типа str. В этом режиме в Windows при чтении символ \r будет удален, а при записи, наоборот, добавлен.

filename = 'w.txt'

f = open(filename, 'r')

print(f.readlines())

f.close()

OUT: ['123\n', '456\n', '789']

***27. Использование инструкции with … as на примере работы с файлами.***

Python поддерживает протокол менеджеров контекста. Этот протокол гарантирует выполнение завершающих действий (например, закрытие файла) вне зависимости от того, произошло исключение внутри блока кода или нет.

Для работы с протоколом предназначена инструкция with ... as. Инструкция имеет следующий формат:

with <Выражение1>[ as <Переменная>] [, ... ,

<ВыражениеN>[ as <Переменная>]]:

<Блок, в котором перехватаем исключения>

Вначале вычисляется <Выражение1>, которое должно возвращать объект, поддерживающий протокол. Этот объект должен иметь два метода: \_\_enter\_\_() и \_\_exit\_\_(). Метод \_\_enter\_\_() вызывается после создания объекта. Значение, возвращаемое этим методом, присваивается переменной, указанной после ключевого слова as. Далее выполняются инструкции внутри тела инструкции with. Если при выполнении возникло исключение, то управление передается методу \_\_exit\_\_(). Если при выполнении инструкций, расположенных внутри тела инструкции with, исключение не возникло, то управление все равно передается методу \_\_exit\_\_().

.write (<Данные>) - записывает строку или последовательность байтов в файл. Если в качестве параметра указана строка, то файл должен быть открыт в текстовом режиме. Для записи последовательности байтов необходимо открыть файл в бинарном режиме.

.writelines(<Последовательность>) - записывает последовательность в файл. Если все элементы последовательности являются строками, то файл должен быть открыт в текстовом режиме. Если все элементы являются последовательностями байтов, то файл должен быть открыт в бинарном режиме.

with open("new\_file7.txt", "w", encoding="utf-8") as f:

f.writelines(["Строка1\n", "Строка2"])

with open("new\_file7.txt", "r", encoding="utf-8") as f:

print(f.read())

OUT:

Строка1

Строка2

Запись в файл можно призводить при помощи функции print(), передав файл в необязательный атрибут file:

with open("new\_file8.txt", "w", encoding="cp1251") as f:

print('Строка1\nСтрока2', file=f)

print('Строка3', end='', file=f) # значение по умолчанию end='\n'

print(' продолжение строки3', file=f)

v = 4

print(f'Простой способ записать в файл f-строку v={v} .', file=f)

with open("new\_file8.txt", "r", encoding="cp1251") as f:

print(f.read())

OUT:

Строка1

Строка2

Строка3 продолжение строки3

Простой способ записать в файл f-строку v=4 .

.read( [<Количество>])- считывает данные из файла. Если файл открыт в текстовом режиме, то возвращается строка, а если в бинарном - последовательность байтов. Если параметр не указан, то возвращается содержимое файла от текущей позиции указателя до конца файла:

with open("new\_file7.txt", "r", encoding="utf-8") as f:

print(f.read(4))

OUT: Стро

.readline ( [<Количество>] ) - считывает из файла одну строку при каждом вызове. Если файл открыт в текстовом режиме, то возвращается строка, а если в бинарном - последовательность байтов. Возвращаемая строка включает символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. При достижении конца файла возвращается пустая строка.

with open("new\_file7.txt", "r", encoding="utf-8") as f:

print(f.readline(), end='')

print('----')

print(f.readline())

print('----')

print(f.readline())

OUT:

Строка1

----

Строка2

----

.readlines () - считывает все содержимое файла в список. Каждый элемент списка будет содержать одну строку, включая символ перевода строки. Исключением является последняя строка. Если она не завершается символом перевода строки, то символ перевода строки добавлен не будет. Если файл открыт в текстовом режиме, то возвращается список строк, а если в бинарном - список объектов типа bytes.

with open("new\_file7.txt", "r", encoding="utf-8") as f:

lines = f.readlines()

lines

OUT: ['Строка1\n', 'Строка2']

.flush () - записывает данные из буфера на диск;

***28. Использование модулей pickle и shelve для сохранения объектов в файл и их восстановления.***

**Pickle**

Модуль pickle предоставляет следующие функции:

* dump(<Объект>, <Файл> [, <Протокол>] [, fix \_ imports=True]) - производит сериализацию объекта и записывает данные в указанный файл. В параметре <Файл> указывается файловый объект, открытый на запись в бинарном режиме.
* load() - читает данные из файла и преобразует их в объект. В параметре <Файл> указывается файловый объект. открытый на чтение в бинарном режиме. Формат функции: load (<Файл> [, fix \_ imports=True] [, encoding="ASCII"] [, errors="strict"])

import pickle # подключаем модуль pickle

obj = ["Строка", (12, 3)] # объект для сохранения

with open('obj1.txt', 'wb') as f:

pickle.dump(obj, f)

with open('obj1.txt', 'rb') as f:

obj\_l = pickle.load(f)

obj\_l

OUT: ['Строка', (12, 3)]

В один файл можно сохранить сразу несколько объектов, последовательно вызывая функцию dump().

Модуль pickle позволяет также преобразовать объект в строку байтов и восстановить объект из строки. Для этого предназначены две функции:

* dumps(<Объект> [, <Протокол>] [, fix\_imports=True]) - производит сериализацию объекта и возвращает последовательность байтов специального формата.
* lоаds(<Последовательность байтов>[, fix\_imports=True] [, errors="strict"])- преобразует последовательность байтов обратно в объект.

**Shelve**

Модуль shelve позволяет сохранять объекты под определенным ключом (задается в виде строки) и предоставляет интерфейс доступа, сходный со словарями. Для сериализации объекта используются возможности модуля pickle, а чтобы записать получившуюся строку по ключу в файл, применяется модуль pickle. Все эти действия модуль shelve производит незаметно для нас.

Чтобы открыть файл с базой объектов, используется функция open(). Функция имеет следующий формат:

open (<Путь к файлу> [, flag="c"] [, protoco1=None] [, writeback=Fa1se])

В необязательном параметре flag можно указать один из режимов открытия файла:

* r - только чтение;
* w - чтение и запись;
* с - чтение и запись (значение по умолчанию). Если файл не существует, он будет создан;
* n - чтение и запись. Если файл не существует, он будет создан. Если файл существует, он будет перезаписан.

Функция open() возвращает объект, с помощью которого производится дальнейшая работа с базой данных. Этот объект имеет следующие методы:

* close() - закрывает файл с базой данных.
* keys() - возвращает объект с ключами;
* values() - возвращает объект to значениями;
* items() - возвращает объект, поддерживающий итерации. На каждой итерации возвращается кортеж, содержащий ключ и значение.
* get(<Ключ> [, <Значение по умолчанию>] ) - если ключ присутствует; то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то возвращается значение None или значение, указанное во втором параметре;
* setdefault(<Ключ> [, <Значение по умолчанию>] ) ---: если ключ присутствует, то метод возвращает значение, соответствующее этому ключу. Если ключ отсутствует, то вставляет новый элемент со значением, указанным во втором параметре, и возвращает это значение. Если второй параметр не указан, значением нового элемента будет None;
* рор (<Ключ> [, <Значение по умолчанию>] ) - удаляет элемент с указанным ключом и возвращает его значение. Если ключ отсутствует, то возвращается значение из второго параметра. Если ключ отсутствует, и второй параметр не указан, то возбуждается исключение KeyError;
* popitem() - удаляет произвольный элемент и возвращает кортеж из ключа и значения. Если файл пустой, возбуждается исключение KeyError;
* clear () - удаляет все элементы. Метод ничего не возвращает в качестве значения;
* update () - добавляет элементы. Метод изменяет текущий объект и ничего не возвращает. Если элемент с указанным ключом уже присутствует, то его значение будет перезаписано.

Помимо этих методов можно воспользоваться функцией len() для получения количества элементов и оператором del для удаления определённого элемента, а также оператором in для проверки существования ключа.

import shelve

db = shelve.open("shl\_1")

db["obj1"] = [1, 2, 3, 4, 5]

db["obj2"] = (6, 7, 8, 9, 10)

db.close()

db = shelve.open('shl\_1')

list(db.keys())

OUT: ['obj2', 'obj1']

list(db.values())

OUT: [(6, 7, 8, 9, 10), [1, 2, 3, 4, 5]]

for k, v in db.items():

print('key: {}, value: {}'.format(k, v))

OUT:

key: obj2, value: (6, 7, 8, 9, 10)

key: obj1, value: [1, 2, 3, 4, 5]

***29. Модули в Python и их отличие от скриптов Python. Варианты синтаксиса импорта модуля и объектов модуля. Применение импортированных объектов. Порядок поиска модулей и специфика их загрузки. Загрузка модулей из глобального репозитория.***

Модуль в языке Python - это обычный файл с расширением .ру. Модуль может содержать любой программный код на языке Python.

Каждая программа, которую мы писали до сих пор, находилась в отдельном файле .ру, который можно считать не только программой, но и модулем. Основное различие между модулем и программой состоит в том, что программа предназначена для того, чтобы ее запускали, тогда как модуль предназначен для того, чтобы его импортировали и использовали в программах.

Не все модули располагаются в файлах с расширением .ру, например, модуль sys встроен в Python, а некоторые модули написаны на других языках программирования (чаще всего на языке С). Однако большая часть библиотеки языка Python написана именно на языке Python.

Импортирование может выполняться несколькими синтаксическими конструкциями, например:

import importable

import importable1, importable2 importableN

import importable as preferred\_name

ПРИМЕР

Одновременный импорт нескольких модулей: import math, collections

Для импортируемого модуля задается произвольное имя: import numpy as np

Использование импортированных модулей: math.cos(0.0)

Обычно все инструкции import помещаются в начало файла .ру после описания модуля. Рекомендуется сначала импортировать модули стандартной библиотеки, затем модули сторонних разработчиков и в последнюю очередь свои собственные модули.

from importable import object as preferred\_name

from importable import objecth object2 objectN

from importable import (objectl, object2, object3, object4, objects,

object6 objectN)

from importable import \*

При использовании конструкции from importable import \* импортируются все объекты из модуля, за исключением тех, чьи имена начинаются с символа подчеркивания. Либо, если в модуле определена глобальная переменная \_all\_ со списком имен, будут импортированы все объекты, имена которых перечислены в переменной \_all\_.

filename = 'C:\Python\Ipynb\2016\lec\_6'

import os.path as p

print(p.basename(filename))

ИЛИ

from os import path

print(path.basename(filename))

OUT: lec\_6

**Порядок поиска файлов, содержащих модули**

Порядок следующий:

1. каталог, где находится сама программа, даже если она вызывается из другого каталога;
2. пути к каталогам из переменной окружения PYTHONPATH, если она определена;
3. пути к каталогам стандартной библиотеки языка Python - они определяются на этапе установки Python.

Программа может импортировать некоторые модули, которые в свою очередь импортируют другие модули, включая те, что уже были импортированы. Это не является проблемой. Всякий раз, когда выполняется попытка импортировать модуль, интерпретатор Python сначала проверяет - не был ли импортирован требуемый модуль ранее.

Когда интерпретатору требуется скомпилированный байт-код модуля, он генерирует его автоматически - этим Python отличается от таких языков программирования, как Java, где компилирование в байт-код должно выполняться явно. Сначала интерпретатор попытается отыскать файл, имя которого совпадает с именем файла, имеющего расширение .ру, но имеющий расширение .руо - это оптимизированный байт-код скомпилированной версии модуля. Если файл с расширением .руо не будет найден (или он более старый, чем файл с расширением .ру), интерпретатор попытается отыскать одноименный файл с расширением .рус - это неоптимизированный байт-код скомпилированной версии модуля. Если интерпретатор обнаружит актуальную скомпилированную версию модуля, он загрузит ее; в противном случае Python загрузит файл с расширением .ру и скомпилирует его в байт-код. В любом случае интерпретатор загрузит в память модуль в виде скомпилированного байт-кода.

!!!При любых последующих попытках импортировать этот модуль интерпретатор будет обнаруживать, что модуль уже импортирован и не будет выполнять никаких действий.

**Установка модулей из глобального репозитария**

pip - это система управления пакетами, которая используется для установки и управления программными пакетами, написанными на Python. Начиная с Python версии 3.4, pip поставляется вместе с интерпретатором python.

pip очень легко использовать для загрузки модулей из PyPI - the Python Package Index.

Для установки пакетов в командной строке нужно выполнить (для выполнения команды могут потребоваться права администратора):

pip install package\_name

pip help - помощь по доступным командам.

pip install package\_name - установка пакета(ов).

pip uninstall package\_name - удаление пакета(ов).

pip list - список установленных пакетов.

pip show package\_name - показывает информацию об установленном пакете.

pip search - поиск пакетов по имени.

pip --proxy user:[passwd@proxy.server](mailto:passwd@proxy.server):port - использование с прокси.

pip install -U - обновление пакета(ов).

pip install --force-reinstall - при обновлении, переустановить пакет, даже если он последней версии.

***30. Импорт кода из пакетов. Организация пакетов в Python.***

Пакет - это простой каталог, содержащий множество модулей и файл с именем \_init\_.ру.

Например, допустим, что у нас имеется некоторое множество файлов модулей, предназначенных для чтения и записи графических файлов различных форматов. Если поместить эти модули в каталог Graphics вместе с пустым файлом \_init\_.ру, то этот каталог превратится в пакет:

`Graphics/

\_\_init\_\_.ру

Bmp.py

Jpeg.py

Png.py

Tiff.py`

Пока каталог Graphics является подкаталогом каталога с программой или находится в пути поиска Python, мы будем иметь возможность импортировать любой из этих модулей и использовать их. Мы должны сделать все возможное, чтобы гарантировать несовпадение имени нашего модуля верхнего уровня (Graphics) с каким-либо из имен верхнего уровня в стандартной библиотеке - с целью избежать конфликтов имен.

import Graphics.Bmp

image = Graphics.Bmp.g\_load("bashful.bmp")

OUT: file bashful.bmp loaded as BMP v2

В некоторых ситуациях бывает удобно загружать все модули пакета одной инструкцией. Для этого необходимо в файле \_*init\_*.py необходимо задать переменную \_*all\_* , которая указывала бы, какие модули должны загружаться.

Например, ниже приводится необходимая строка для файла Graphics/\_*init\_*.py:

all = ["Bmp", "Jpeg", "Png", "Tiff"]

Этим ограничивается необходимое содержимое файла, но помимо этого, мы можем поместить в него любой программный код, какой только пожелаем.

Теперь мы можем использовать другую разновидность инструкции import:

Синтаксис from package import \* напрямую импортирует все имена модулей, упомянутые в списке all .

from Graphics import \*