1. В чем разница между списком и кортежем?

Мне задавали этот вопрос буквально на каждом собеседовании по Python/data science. Выучите ответ как свои пять пальцев:

1.Список можно изменить после создания.

2.Кортеж нельзя изменить после создания.

3.Список упорядочен. Он представляет собой упорядоченные последовательности объектов, как правило, одного и того же типа. Например, все имена пользователей упорядочены по дате создания: `["Seth", "Ema", "Eli"]`.

4.У кортежа есть структура. В каждом индексе могут сосуществовать различные типы данных. Например, такая запись базы данных в памяти: `(2, "Ema", "2020-04-16") # id, name, created_at`.

2. Как выполняется интерполяция строк?

Без импорта класса `Template` есть три способа интерполяции строк:

```
name = 'Chris'
# 1. f strings
print(f'Hello {name}')
# 2. % operator
print('Hey %s %s' % (name, name))
# 3. format
print(
 "My name is {}".format((name))
)
```

3. В чем разница между is и ==?

Когда я был начинающим разработчиком, то не видел разницы… привет, баги. Так что для протокола: `is` проверяет идентичность, а `==` проверяет равенство.

Рассмотрим пример. Создайте несколько списков и назначьте им имена. Обратите внимание, что ниже `b` указывает на тот же объект, что и `a`:

```
a = [1,2,3]
b = a
c = [1,2,3]
```

Проверьте равенство и обратите внимание, что все объекты равны:

```
print(a == b)
print(a == c)
#=> True
#=> True
```

Но являются ли все они идентичными? Нет:

```
print(a is b)
print(a is c)
#=> True
#=> False
```

Можем проверить это, распечатав их идентификаторы объектов:

```
print(id(a))
print(id(b))
print(id(c))
#=> 4369567560
#=> 4369567560
#=> 4369567624
```

Идентификатор `c` отличается от идентификатора `a` и `b`.

4. Что такое декоратор?

Еще один вопрос, который мне задавали на каждом собеседовании. Тема заслуживает отдельной статьи, но для базовой подготовки достаточно просто написать собственный пример.

Декоратор позволяет добавить новую функциональность к существующей функции. Это делается следующим образом. Функция передается декоратору, а он выполняет и существующий, и дополнительный код.

Напишем декоратор, который записывает в журнал вызовы другой функции.

Напишите функцию декоратора. В качестве аргумента он принимает функцию `func`. Декоратор определяет функцию `log_function_called`, которая вызывает `func()` и выполняет некоторый код `print(f'{func} called.')`. Затем возвращает определенную им функцию:

```
def logging(func):
 def log_function_called():
   print(f'{func} called.')
   func()
 return log_function_called
```

Напишем другие функции, к которым добавим декоратор (потом, не сейчас):

```
def my_name():
  print('chris')
def friends_name():
  print('naruto')
my_name()
friends_name()
#=> chris
#=> naruto
```

Теперь добавим декоратор к ним обоим:

```
@logging
def my_name():
```

```
    print('chris')
@logging
def friends_name():
    print('naruto')
my_name()
friends_name()
#=> <function my_name at 0x10fca5a60> called.
#=> chris
#=> <function friends_name at 0x10fca5f28> called.
#=> naruto
```

Теперь легко добавить ведение журнала в любую функцию, которую мы пишем. Достаточно написать перед ней `@logging`.

5. Объясните функцию range

Range генерирует список целых чисел. Ее можно использовать тремя способами.

Функция принимает от одного до трех аргументов. Обратите внимание, что я завернул каждый пример в список, чтобы видеть генерируемые значения.

`range(stop)` — генерирует целые числа от 0 до целого числа `stop`:

```
[i for i in range(10)]
#=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`range(start, stop)` — генерирует целые числа от `start` до `stop`:

```
[i for i in range(2,10)]
#=> [2, 3, 4, 5, 6, 7, 8, 9]
```

`range(start, stop, step)` — генерирует целые числа от `start` до `stop` с интервалами step:

```
[i for i in range(2,10,2)]
#=> [2, 4, 6, 8]
```

Серж Боремчук предложил более подходящий способ:

```
list(range(2,10,2))
#=> [2, 4, 6, 8]
```

6. Определите класс car с двумя атрибутами: color и speed. Затем создайте экземпляр и верните speed

Вот как это сделать:

```
class Car :
    def __init__(self, color, speed):
        self.color = color
        self.speed = speed
car = Car('red','100mph')
car.speed
#=> '100mph'
```

7. В чем разница между методами экземпляра, класса и статическими методами в Python?

Методы экземпляра: принимают параметр `self` и относятся к определенному экземпляру класса.

Статические методы: используют декоратор `@staticmethod`, не связаны с конкретным экземпляром и являются автономными (атрибуты класса или экземпляра не изменяются).

Методы класса: принимают параметр `cls`, можно изменить сам класс.

Проиллюстрируем разницу на вымышленном классе `CoffeeShop`:

```
class CoffeeShop:
    specialty = 'espresso'
    def __init__(self, coffee_price):
        self.coffee_price = coffee_price

    # instance method
    def make_coffee(self):
        print(f'Making {self.specialty} for ${self.coffee_price}')

    # static method
    @staticmethod
    def check_weather():
        print('Its sunny')
    # class method
    @classmethod
    def change_specialty(cls, specialty):
        cls.specialty = specialty
        print(f'Specialty changed to {specialty}')
```

У класса `CoffeeShop` есть атрибут `specialty` (фирменный напиток), установленный по умолчанию в значение `'espresso'`. Каждый экземпляр `CoffeeShop` инициализируется с атрибутом `coffee_price`. У него также три метода: метод экземпляра, статический метод и метод класса.

Давайте инициализируем экземпляр с атрибутом `coffee_price`, равным `5`. Затем вызовем метод экземпляра `make_coffee`:

```
coffee_shop = CoffeeShop('5')
coffee_shop.make_coffee()
#=> Making espresso for $5
```

Теперь вызовем статический метод. Статические методы не могут изменять состояние класса или экземпляра, поэтому обычно используются для служебных функций, например, сложения двух чисел. Наши проверяют погоду. Говорят, что солнечно. Отлично!

```
coffee_shop.check_weather()
```

```
#=> Its sunny
```

Теперь используем метод класса для изменения фирменного напитка (`specialty`), а затем сделаем кофе (`make_coffee`):

```
coffee_shop.change_specialty('drip coffee')
#=> Specialty changed to drip coffee
coffee_shop.make_coffee()
#=> Making drip coffee for $5
```

Обратите внимание, что `make_coffee` раньше делал эспрессо, а теперь заваривает капельную кофеварку (`drip coffee`).

8. В чем разница между func и func()?

Вопрос должен проверить ваше понимание, что все функции в Python также являются объектами:

```
def func():
    print('Im a function')
func
#=> function __main__.func>
func()
#=> Im a function
```

`func` — это представляющий функцию объект, который можно назначить переменной или передать другой функции. Функция `func()` с круглыми скобками вызывает функцию и возвращает результат.

9. Объясните, как работает функция map

Она возвращает объект (итератор), который перебирает значения, применяя функцию к каждому элементу. В случае необходимости объект можно преобразовать в список:

```
def add_three(x):
```

```
    return x + 3
li = [1,2,3]
list(map(add_three, li))
#=> [4, 5, 6]
```

Здесь к каждому элементу в списке мы добавляем число 3.

10. Объясните, как работает функция reduce

Это может быть сложновато сразу понять, пока вы не используете ее несколько раз.

`reduce` принимает функцию и последовательность — и проходит по этой последовательности. На каждой итерации в функцию передаются как текущий элемент, так и выходные данные предыдущего элемента. В конце концов, возвращается одно значение:

```
from functools import reduce
def add_three(x,y):
    return x + y
li = [1,2,3,5]
reduce(add_three, li)
#=> 11
```

Возвращается 11, что является суммой 1+2+3+5.

11. Объясните, как работает функция filter

Функция делает буквально то, о чем говорит ее название: она фильтрует элементы в последовательности.

Каждый элемент передается функции, которая включает его в последовательность, если по условию получает `True`, и отбрасывает в случае `False`:

```
def add_three(x):
    if x % 2 == 0:
        return True
```

```
    else:
        return False
li = [1,2,3,4,5,6,7,8]
[i for i in filter(add_three, li)]
#=> [2, 4, 6, 8]
```

Обратите внимание, как удалены все элементы, которые не делятся на 2.

12. Переменные в Python передаются по ссылке или по значению?

Будьте готовы спуститься в кроличью нору семантики, если загуглите этот вопрос и прочтете несколько первых страниц.

В общем, все имена передаются по ссылке, но в некоторых ячейках памяти хранятся объекты, а в других — указатели на другие ячейки памяти.

```
name = 'object'
```

Давайте посмотрим, как это работает со строками. Создадим экземпляр имени и объекта, на который указывают другие имена. Затем удалим первое:

```
x = 'some text'
y = x
x is y #=> True
del x # удаляем имя 'a' , но не объект в памяти
z = y
y is z #=> True
```

Мы видим, что все имена указывают на один и тот же объект в памяти, который остался нетронутым после операции удаления имени `del x`.

Вот еще один интересный пример с функцией:

```
name = 'text'
def add_chars(str1):
```

```
    print( id(str1) ) #=> 4353702856
    print( id(name) ) #=> 4353702856

    # новое имя, тот же объект
    str2 = str1

    # создаем новое имя (не отличается от предыдущего) и новый объект
    str1 += 's'
    print( id(str1) ) #=> 4387143328

    # объект не изменился
    print( id(str2) ) #=> 4353702856

add_chars(name)
print(name) #=>text
```

Обратите внимание, что добавление буквы `s` в строку внутри функции создает новое имя — и новый объект тоже. Даже если у нового объекта то же самое имя, что и у существующего.

13. Как развернуть список?

Обратите внимание, что `reverse()` вызывается в списке и изменяет его. Сама функция не возвращает измененный список:

```
li = ['a','b','c']
print(li)
li.reverse()
print(li)
#=> ['a', 'b', 'c']
#=> ['c', 'b', 'a']
```

14. Как работает умножение строк?

Посмотрим результат умножения строки `'cat'` на `3`:

```
'cat' * 3
#=> 'catcatcat'
```

В результате содержимое строки повторяется трижды.

15. Как работает умножение списка?

Посмотрим на результат умножения списка `[1,2,3]` на 2:

```
[1,2,3] * 2
#=> [1, 2, 3, 1, 2, 3]
```

Содержание списка `[1,2,3]` повторяется дважды.

16. Что означает self в классе?

Self ссылается на экземпляр класса. Так метод может обновлять объект, к которому принадлежит.

Ниже передача `self` в `__init__()` дает возможность установить цвет экземпляра при инициализации:

```
class Shirt:
    def __init__(self, color):
        self.color = color

s = Shirt('yellow')
s.color
#=> 'yellow'
```

17. Как объединить списки в Python?

Списки объединяются при сложении. Обратите внимание, что с массивами так не получается:

```
a = [1,2]
b = [3,4,5]
a + b
#=> [1, 2, 3, 4, 5]
```

18. В чем разница между глубокой и мелкой копиями?

Обсудим это в контексте изменяемого объекта — списка. Для неизменяемых объектов глубокое и мелкое (поверхностное) копирование обычно не отличаются.

Рассмотрим три сценария.

I) Поставьте ссылку на исходный объект. Она отсылает новое имя `li2` к тому же месту в памяти, на которое указывает `li1`. Поэтому любое изменение в `li1` также происходит с `li2`:

```
li1 = [['a'],['b'],['c']]
li2 = li1
li1.append(['d'])
print(li2)
#=> [['a'], ['b'], ['c'], ['d']]
```

II) Создайте мелкую копию оригинала. Ее можно создать с помощью конструктора `list()` или `mylist.copy()`.

Мелкая копия создает новый объект, но заполняет его ссылками на оригинал. Таким образом, добавление нового объекта в исходный список `li3` не отразится в `li4`, а вот изменение объектов в `li3` — отразится:

```
li3 = [['a'],['b'],['c']]
li4 = list(li3)
li3.append([4])
print(li4)
#=> [['a'], ['b'], ['c']]
li3[0][0] = ['X']
print(li4)
```

```
#=> [[['X']], ['b'], ['c']]
```

III) Создайте глубокую копию. Это делается с помощью `copy.deepcopy()`. Оригинал и копия полностью независимы, а изменения в одном не оказывают никакого влияния на другой:

```
import copy
li5 = [['a'],['b'],['c']]
li6 = copy.deepcopy(li5)
li5.append([4])
li5[0][0] = ['X']
print(li6)
#=> [['a'], ['b'], ['c']]
```

19. В чем разница между списками и массивами?

Примечание: в стандартной библиотеке Python есть объект array, но здесь мы специально обсуждаем массив из популярной библиотеки Numpy.

Списки в каждом индексе можно заполнять разными типами данных. Массивы требуют однородных элементов.

Арифметические действия в списках добавляют или удаляют элементы из списка. Арифметические действия на массивах соответствуют функциям линейной алгебры.

Массивы используют меньше памяти и обладают значительно большей функциональностью.

20. Как объединить два массива?

Помните, что массивы — это не списки. Это библиотека Numpy и здесь работает линейная алгебра.

Для объединения массивов нужно использовать соответствующую функцию Numpy:

```
import numpy as np
```

```
a = np.array([1,2,3])
b = np.array([4,5,6])
np.concatenate((a,b))
#=> array([1, 2, 3, 4, 5, 6])
```

21. Что вам нравится в Python?

Примечание: это очень субъективный вопрос, и логично адаптировать ответ в зависимости от того, на какую должность вы претендуете.

Python очень удобочитаем, и есть так называемый «питоновский способ» решения почти любой задачи, то есть самый понятный, ясный и лаконичный код.

Это мне кажется противоположностью Ruby, где часто много способов решить задачу без четких указаний, какой вариант предпочтительнее.

22. Какая ваша любимая библиотека в Python?

Примечание: это тоже субъективно, см. вопрос 21.

При работе с большим количеством данных трудно найти что-то полезнее, чем pandas. С этой библиотекой обработка и визуализация данных становятся проще простого.

23. Назовите изменяемые и неизменяемые объекты

Неизменяемость означает, что состояние нельзя изменить после создания. Примеры: int, float, bool, string и tuple.

Состояние изменяемых объектов можно изменить. Примеры: list, dict и set.

24. Как округлить число до трех десятичных знаков?

Используйте функцию `round(value, decimal_places)`:

```
a = 5.12345
round(a,3)
#=> 5.123
```

25. Как разбить список?

Синтаксис функции включает три аргумента: `list[start:stop:step]`, где `step` — это интервал, через который возвращаются элементы:

```
a = [0,1,2,3,4,5,6,7,8,9]
print(a[:2])
#=> [0, 1]
print(a[8:])
#=> [8, 9]
print(a[2:8])
#=> [2, 3, 4, 5, 6, 7]
print(a[2:8:2])
#=> [2, 4, 6]
```

26. Что такое pickle?

Pickle — это модуль сериализации и десериализации объектов в Python.

В примере ниже мы сериализуем и десериализуем список словарей:

```
import pickle
obj = [
    {'id':1, 'name':'Stuffy'},
    {'id':2, 'name': 'Fluffy'}
]
with open('file.p', 'wb') as f:
    pickle.dump(obj, f)
with open('file.p', 'rb') as f:
```

```
    loaded_obj = pickle.load(f)
print(loaded_obj)
#=> [{'id': 1, 'name': 'Stuffy'}, {'id': 2, 'name': 'Fluffy'}]
```

27. Какая разница между словарями и JSON?

Dict (словарь) — это тип данных Python, представляющий собой набор индексированных, но неупорядоченных пар ключ-значение.

JSON — просто строка, которая следует заданному формату и предназначена для передачи данных.

28. Какие ORM вы использовали в Python?

Технология ORM (object-relational mapping, объектно-реляционное отображение) связывает модели данных (обычно в приложении) с таблицами БД и упрощает транзакции с базой данных.

В контексте Flask обычно используется SQLAlchemy, а у Django собственная ORM.

29. Как работают any() и all()?

Any возвращает `true`, если хоть один элемент в последовательности соответствует условию, то есть является `true`.

All возвращает `true` только в том случае, если условию соответствуют все элементы в последовательности.

```
a = [False, False, False]
b = [True, False, False]
c = [True, True, True]
print( any(a) )
print( any(b) )
print( any(c) )
#=> False
```

```
#=> True
#=> True
print( all(a) )
print( all(b) )
print( all(c) )
#=> False
#=> False
#=> True
```

30. Где быстрее поиск: в словарях или списках?

Поиск значения в списке занимает O(n) времени, потому что нужно пройти весь список.

Поиск ключа в словаре занимает O(1) времени, потому что это хэш-таблица.

Разница во времени может быть огромной, если значений много, поэтому для производительности обычно рекомендуют словари. Но у них есть другие ограничения, такие как необходимость уникальных ключей.

31. В чем разница между модулем и пакетом?

Модуль — это файл или набор файлов, которые импортируются вместе:

```
import sklearn
```

Пакет — это каталог с модулями:

```
from sklearn import cross_validation
```

Таким образом, пакеты — это модули, но не все модули являются пакетами.