Топ 100 вопросов по Python

2023-02-17

Содержание

Junior

1. Что такое Python? Какие преимущества использования Python?

Python - это высокоуровневый интерпретируемый язык программирования общего назначения. Будучи языком общего назначения, он может быть использован для создания практически любого типа приложений при наличии соответствующих инструментов/библиотек.

Кроме того, python поддерживает объекты, модули, потоки, обработку исключений и автоматическое управление памятью, что помогает моделировать реальные проблемы и создавать приложения для решения этих проблем.

Преимущества использования Python:

Python - это язык программирования общего назначения, который имеет простой, легко изучаемый синтаксис, подчеркивающий удобочитаемость и, следовательно, снижающий затраты на сопровождение программ. Более того, язык способен выполнять сценарии, является полностью открытым и поддерживает пакеты сторонних разработчиков, что способствует модульности и повторному использованию кода.

Его высокоуровневые структуры данных в сочетании с динамической типизацией и динамическим связыванием привлекают огромное сообщество разработчиков для быстрой разработки и развертывания приложений.

2. Что такое динамически типизированный язык?

Прежде чем понять, что такое динамически типизированный язык, мы должны узнать, что такое типизация. Типизация относится к проверке типов в языках программирования. В языке с сильной типизацией, таком как Python, “1” + 2 приведет к ошибке типа, поскольку эти языки не допускают “приведения типов” (неявного преобразования типов данных). С другой стороны, слабо типизированный язык, такой как JavaScript, просто выведет “12” в качестве результата.

Проверка типов может быть выполнена на двух этапах:

1. Статический - типы данных проверяются перед выполнением.
2. Динамический - типы данных проверяются во время выполнения.

Python - интерпретируемый язык, каждый оператор выполняется построчно, поэтому проверка типов выполняется на лету, во время выполнения. Следовательно, Python является динамически типизированным языком.

3. Что такое интерпретируемый язык?

Интерпретированный язык выполняет свои утверждения построчно. Такие языки, как Python, JavaScript, R, PHP и Ruby, являются яркими примерами интерпретируемых языков. Программы, написанные на интерпретируемом языке, выполняются непосредственно из исходного кода, без промежуточного этапа компиляции.

4. Что такое PEP 8 и почему он важен?

PEP расшифровывается как Python Enhancement Proposal. PEP - это официальный проектный документ, предоставляющий информацию сообществу Python или описывающий новую функцию для Python или его процессов.

PEP 8 особенно важен, поскольку в нем документированы руководящие принципы стиля для кода Python. Очевидно, что вклад в сообщество разработчиков открытого кода Python требует от вас искреннего и строгого следования этим руководящим принципам стиля.

5. Что такое область видимости в Python?

Каждый объект в Python функционирует в пределах области видимости. **Область видимости** - это блок кода, в котором объект в Python остается актуальным. Пространства имен однозначно идентифицируют все объекты внутри программы.

В Python существует **3 области видимости**:

1. Локальная
2. Глобальная
3. Нелокальная

Однако эти пространства имен также имеют область видимости, определенную для них, где вы можете использовать их объекты без префикса. Ниже приведено несколько примеров областей видимости, создаваемых во время выполнения кода в Python:

Локальная область видимости относится к локальным объектам, доступным в текущей функции.

Локальная область видимости - определенная внутри функции, метода или выражения. Переменные, определенные внутри этой области, недоступны за ее пределами.

```
x = 10

def my_func(a, b):
    print(x)
    print(z)

>>>my_func(1, 2)

10
Traceback (most recent call last):
    File "<pyshell#19>", line 1, in <module>
        my_func(1, 2)
    File "<pyshell#18>", line 3, in my_func
        print(z)
NameError: name 'z' is not defined
```

Глобальная область видимости относится к объектам, доступным во время выполнения кода с момента их создания.

Глобальная область видимости - определенная вне функций, методов и выражений. Переменные, определенные в глобальной области видимости, доступны везде в коде.

Область видимости на уровне модуля относится к глобальным объектам текущего модуля, доступным в программе.

Глобальная область видимости относится ко всем встроенным именам, вызываемым в программе. Объекты в этой области видимости ищутся в последнюю очередь, чтобы найти имя, на которое ссылаются.

```
def my_func(a, b):
    global x
    print(x)
    x = 5
    print(x)

if __name__ == '__main__':
    x = 10
    my_func(1, 2)
    print(x)

10
5
5
```

 Примечание: Объекты локальной области видимости могут быть синхронизированы с объектами глобальной области видимости с помощью таких ключевых слов, как **global**.

В Python 3 было добавлено новое ключевое слово под названием `nonlocal`. С его помощью мы можем добавлять переопределение области во внутреннюю область. Вы можете ознакомиться со всей необходимой на данный счет информацией в PEP 3104. Это наглядно демонстрируется в нескольких примерах. Один из самых простых – это создание функции, которая может увеличиваться:

```
def counter():
    num = 0
    def incrementer():
        num += 1
        return num
    return incrementer
```

Если вы попробуете запустить этот код, вы получите ошибку UnboundLocalError, так как переменная num ссылается прежде, чем она будет назначена в самой внутренней функции. Давайте добавим nonlocal в наш код:

```
def counter():
    num = 0
    def incrementer():
        nonlocal num
        num += 1
        return num
    return incrementer

c = counter()
print(c) # <function counter.<locals>.incrementer at 0x7f45caf44048>

c() # 1
c() # 2
c() # 3
```

6. Что такое списки и кортежи? В чем ключевое различие между ними?

Списки и кортежи - это типы данных последовательности, которые могут хранить коллекцию объектов в Python. Объекты, хранящиеся в обеих последовательностях, могут иметь различные типы данных. Списки представлены квадратными скобками `['sara', 6, 0.19]`, а кортежи - круглыми `('ansh', 5, 0.97)`.

Но в чем реальная разница между ними? Ключевое различие между ними заключается в том, что списки являются изменяемыми, а кортежи, напротив, неизменяемыми объектами. Это означает, что списки можно

изменять, добавлять или нарезать на ходу, а кортежи остаются неизменными и не могут быть изменены никаким образом. Вы можете выполнить следующий пример, чтобы убедиться в разнице:

```
my_tuple = ('sara', 6, 5, 0.97)
my_list = ['sara', 6, 5, 0.97]
print(my_tuple[0])      # output => 'sara'
print(my_list[0])       # output => 'sara'
my_tuple[0] = 'ansh'    # modifying tuple => throws an error
my_list[0] = 'ansh'     # modifying list => list modified
print(my_tuple[0])      # output => 'sara'
print(my_list[0])       # output => 'ansh'
```

7. Каковы общие встроенные типы данных в Python?

В Python существует несколько встроенных типов данных. Хотя Python не требует явного определения типов данных при объявлении переменных, ошибки могут возникнуть, если пренебречь знанием типов данных и их совместимости друг с другом. Python предоставляет функции type() и isinstance() для проверки типа этих переменных. Эти типы данных можно сгруппировать в следующие категории-

Тип None:

Ключевое слово **None** представляет нулевые значения в Python. Операция булева равенства может быть выполнена с использованием этих объектов **NoneType**.

NoneType Представляет значения `NULL` в Python.

Числовые типы:

Существует три различных числовых типа - целые числа (integers), числа с плавающей точкой (floating-point) и комплексные числа(complex numbers). Кроме того, булевы числа являются подтипом целых чисел.

- **int** Хранит целочисленные литералы, включая шестнадцатеричные, восьмеричные и двоичные числа, как целые числа
- **float** Хранит литералы, содержащие десятичные значения и/или знаки экспоненты, как числа с плавающей точкой
- **complex** Хранит комплексные числа в виде `(A + Bj)` и имеет атрибуты: real и imag
- **bool** Хранит булево значение (True или False).

Типы последовательностей:

Согласно Python Docs, существует три основных типа последовательностей - списки (lists), кортежи (tuples) и объекты диапазона (range objects). Типы последовательностей имеют операторы `in` и `not in`, определенные для обхода их элементов. Эти операторы имеют тот же приоритет, что и операции сравнения.

- **list** Неизменяемая последовательность, используемая для хранения коллекции элементов.
- **tuple** Неизменяемая последовательность, используемая для хранения коллекции элементов.
- **range** Представляет собой неизменяемую последовательность чисел, генерируемую во время выполнения.
- **str** Неизменяемая последовательность кодовых точек Unicode для хранения текстовых данных.

Стандартная библиотека также включает дополнительные типы для обработки:

1. Двоичные данные
2. Текстовые строки, такие как `str`.

Тип словарь (dict):

Объект отображения может отображать хэшируемые значения на произвольные объекты в Python. Объекты отображения являются изменяемыми, и в настоящее время существует только один стандартный тип отображения - **dict**.

- **dict** Хранит список пар ключ: значение, разделенных запятыми.

Типы множеств:
В настоящее время в Python есть два встроенных типа множеств - **set** и **frozenset**.
Тип **set** является изменяемым и поддерживает такие методы, как `add()` и `remove()`.
Тип **frozenset** является неизменяемым и не может быть изменен после создания.

- **set** Мутабельная неупорядоченная коллекция отдельных хэшируемых объектов.
- **frozenset** Неизменяемая коллекция отдельных хэшируемых объектов.

set является изменяемым и поэтому не может быть использован в качестве ключа словаря. С другой стороны, **frozenset** является неизменяемым и, следовательно, хэшируемым, и может использоваться как ключ словаря или как элемент другого множества.

Модули:

Module - это дополнительный встроенный тип, поддерживаемый интерпретатором Python. Он поддерживает одну специальную операцию, т.е. доступ к атрибуту: `mymod.myobj`, где `mymod` - модуль, а `myobj` ссылается на имя, определенное в модуле.

Таблица символов модуля находится в специальном атрибуте модуля **dict**, но прямое присвоение этому модулю невозможно и не рекомендуется.

Типы Callable:

Callable типы - это типы, к которым может быть применен вызов функции. Это могут быть определяемые пользователем функции, методы экземпляра, функции генератора и некоторые другие встроенные функции, методы и классы.

8. Что такое pass в Python?

Ключевое слово **pass** представляет собой нулевую операцию в Python. Обычно оно используется для заполнения пустых блоков кода, который может выполняться во время исполнения, но еще не написан. Без оператора **pass** в следующем коде мы можем столкнуться с некоторыми ошибками во время выполнения кода.

```
def myEmptyFunc():
    # do nothing
    pass
myEmptyFunc()    # nothing happens
## Without the pass keyword
# File "<stdin>", line 3
# IndentationError: expected an indented block
```

9. Что такое модули и пакеты в Python?

Пакеты Python и модули Python - это два механизма, которые позволяют осуществлять модульное программировани в Python. Модулирование имеет несколько преимуществ:

- Простота: Работа над одним модулем помогает сосредоточиться на относительно небольшой части решаемой задачи. Это делает разработку более простой и менее подверженной ошибкам.
- Удобство обслуживания: Модули предназначены для обеспечения логических границ между различными проблемными областями. Если они написаны таким образом, что уменьшают взаимозависимость, то меньше вероятность того, что изменения в модуле могут повлиять на другие части программы.
- Возможность повторного использования: Функции, определенные в модуле, могут быть легко использованы повторно в других частях приложения.
- Разметка: Модули обычно определяют отдельное пространство имен, что помогает избежать путаницы между идентификаторами из других частей программы.

Модули, в общем случае, это просто файлы Python с расширением `.py`, в которых может быть определен и реализован набор функций, классов или переменных. Они могут быть импортированы и инициализированы один раз с помощью оператора `import`. Если требуется частичная функциональность, импортируйте необходимые классы или функции с помощью оператора `import`: `from foo import bar`.

Пакеты позволяют иерархически структурировать пространство имен модуля с помощью точечной нотации. Как модули помогают избежать столкновений между именами глобальных переменных, так и пакеты помогают избежать столкновений между именами модулей.

Создать пакет очень просто, поскольку он использует присущую системе файловую структуру. Просто поместите модули в папку, и вот оно, имя папки как имя пакета. Для импорта модуля или его содержимого из этого пакета требуется, чтобы имя пакета было префиксом к имени модуля, соединенным точкой.

Примечание: технически вы можете импортировать и пакет, но, увы, это не импортирует модули внутри пакета в локальное пространство имен.

10. Что такое глобальные, защищенные и приватные атрибуты в Python?

Глобальные переменные- это общедоступные переменные, которые определены в глобальной области видимости. Чтобы использовать переменную в глобальной области видимости внутри функции, мы используем ключевое слово **global**.

Защищенные атрибуты (Protected attributes) - это атрибуты, определенные с префиксом подчеркивания к их идентификатору, например, `_sara`. К ним все еще можно получить доступ и изменить их извне класса, в котором они определены, но ответственный разработчик должен воздержаться от этого.

Приватные атрибуты (Private attributes) - это атрибуты с двойным подчеркиванием в префиксе к их идентификатору, например __ansh. Они не могут быть доступны или изменены извне напрямую, и при такой попытке будет выдана ошибка `AttributeError`.

11. Как используется self в Python?

`self` используется для представления экземпляра класса. С помощью этого ключевого слова вы можете получить доступ к атрибутам и методам класса в python.

`self` связывает атрибуты с заданными аргументами. `self` используется в разных местах и часто считается ключевым словом. Но в отличие от C++, `self` не является ключевым словом в Python.

12. Что такое init?

init - это метод-конструктор в Python, который автоматически вызывается для выделения памяти при создании нового объекта/экземпляра. Все классы имеют метод **init**, связанный с ними. Он помогает отличить методы и атрибуты класса от локальных переменных.

```
# class definition
class Student:
    def __init__(self, fname, lname, age, section):
        self.firstname = fname
        self.lastname = lname
        self.age = age
        self.section = section
# creating a new object
stu1 = Student("Sara", "Ansh", 22, "A2")
```

13. Что такое break, continue и pass в Python?

Оператор **break** немедленно завершает цикл, а управление переходит к оператору после тела цикла.

Оператор **continue** завершает текущую итерацию оператора, пропускает остальной код в текущей итерации, а управление переходит к следующей итерации цикла.

Ключевое слово **pass** в Python обычно используется для заполнения пустых блоков и аналогично пустому утверждению, представленному точкой с запятой в таких языках, как Java, C++, Javascript и т.д.

14. Что такое модульные тесты в Python?

Юнит-тесты - это структура модульного тестирования в Python.

Юнит-тестирование означает тестирование различных компонентов программного обеспечения по отдельности. Можете ли вы подумать о том, почему модульное тестирование важно? Представьте себе сценарий: вы создаете программное обеспечение, которое использует три компонента, а именно A, B и C. Теперь предположим, что в какой-то момент ваше программное обеспечение ломается. Как вы определите, какой компонент был ответственен за поломку программы? Может быть, это компонент A вышел из строя, который, в свою очередь, вышел из строя компонент B, что и привело к поломке программного обеспечения. Таких комбинаций может быть множество.

Вот почему необходимо должным образом протестировать каждый компонент, чтобы знать, какой компонент может быть ответственен за сбой программного обеспечения.

15. Что такое docstring в Python?

docstring - это многострочная строка, используемая для документирования определенного участка кода.

В `docstring` должно быть описано, что делает функция или метод.

16. Что такое срез в Python?

Как следует из названия, “срез” - это взятие частей.

Синтаксис следующий [start : stop : step].

- start - начальный индекс, с которого производится нарезка списка или кортежа
- stop - конечный индекс или место нарезки.
- step - количество шагов для перехода.

Значение по умолчанию для start - 0, stop - количество элементов, step - 1.
Срезы можно выполнять для строк, массивов, списков и кортежей.

```
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(numbers[1 : : 2])  #output : [2, 4, 6, 8, 10]
```

17. Объясните, как можно сделать Python Script исполняемым на Unix?

Файл сценария должен начинаться с #!/usr/bin/env python

18. В чем разница между массивами и списками в Python?

Массивы в python могут содержать элементы только одного типа данных, т.е. тип данных массива должен быть однородным. Это тонкая обертка вокруг массивов языка C, и они потребляют гораздо меньше памяти, чем списки.

Списки в python могут содержать элементы разных типов данных, то есть тип данных списков может быть неоднородным. Их недостатком является потребление большого объема памяти.

```
import array
a = array.array('i', [1, 2, 3])
for i in a:
    print(i, end=' ')    #OUTPUT: 1 2 3
a = array.array('i', [1, 2, 'string'])    #OUTPUT: TypeError: an integer is required (got type str)
a = [1, 2, 'string']
for i in a:
   print(i, end=' ')    #OUTPUT: 1 2 string
```

Middle / Senior

19. Как осуществляется управление памятью в Python?

Управление памятью в Python осуществляется менеджером памяти Python. Память, выделяемая менеджером, представляет собой частное пространство кучи, предназначенное для Python. Все объекты Python хранятся в этой куче, и, будучи частной, она недоступна программисту. Тем не менее, Python предоставляет некоторые основные функции API для работы с частным пространством кучи.

Кроме того, Python имеет встроенную сборку мусора для утилизации неиспользуемой памяти для частного пространства кучи.

20. Что такое пространства имен Python? Зачем они используются?

Пространство имен в Python гарантирует, что имена объектов в программе уникальны и могут использоваться без каких-либо конфликтов. Python реализует эти пространства имен в виде словарей, в которых “имя как ключ” сопоставлено с соответствующим “объектом как значением”. Это позволяет нескольким пространствам имен использовать одно и то же имя и сопоставлять его с отдельным объектом. Ниже приведены несколько примеров пространств имен:

Локальное пространство имен включает локальные имена внутри функции. Пространство имен временно создается для вызова функции и очищается после возвращения функции.

Глобальное пространство имен включает имена из различных импортированных пакетов/модулей, которые используются в текущем проекте. Это пространство имен создается при импорте пакета в скрипт и сохраняется до выполнения скрипта.

Встроенное пространство имен включает встроенные функции ядра Python и встроенные имена для различных типов исключений.

Жизненный цикл пространства имен зависит от области видимости объектов, с которыми они сопоставлены. Если область видимости объекта заканчивается, жизненный цикл этого пространства имен завершается. Следовательно, невозможно получить доступ к объектам внутреннего пространства имен из внешнего пространства имен.

21. Что такое разрешение области видимости в Python?

Иногда объекты в одной области видимости имеют одинаковые имена, но функционируют по-разному. В таких случаях разрешение области видимости в Python происходит автоматически. Вот несколько примеров такого поведения:

Модули Python 'math' и 'cmath' имеют множество функций, общих для обоих - log10(), acos(), exp() и т.д. Чтобы разрешить эту двусмысленность, необходимо снабдить их префиксом соответствующего модуля, например, math.exp() и cmath.exp().

Рассмотрим приведенный ниже код, объект temp был инициализирован на 10 глобально и затем на 20 при вызове функции. Однако вызов функции не изменил значение temp глобально. Здесь мы можем заметить, что Python проводит четкую границу между глобальными и локальными переменными, рассматривая их пространства имен как отдельные личности.

```
temp = 10   # global-scope variable
def func():
     temp = 20   # local-scope variable
     print(temp)
print(temp)   # output => 10
func()    # output => 20
print(temp)   # output => 10
```

Это поведение может быть переопределено с помощью ключевого слова global внутри функции, как показано в следующем примере:

```
temp = 10   # global-scope variable
def func():
     global temp
     temp = 20   # local-scope variable
     print(temp)
print(temp)   # output => 10
func()    # output => 20
print(temp)   # output => 20
```

22. Что такое декораторы в Python?

Декораторы в Python - это, по сути, функции, которые добавляют функциональность к существующей функции в Python без изменения структуры самой функции. В Python они обозначаются @decorator_name и вызываются по принципу "снизу вверх". Например:

```
# decorator function to convert to lowercase
def lowercase_decorator(function):
   def wrapper():
       func = function()
       string_lowercase = func.lower()
       return string_lowercase
   return wrapper

# decorator function to split words
def splitter_decorator(function):
   def wrapper():
       func = function()
       string_split = func.split()
       return string_split
   return wrapper

@splitter_decorator # this is executed next
@lowercase_decorator # this is executed first
def hello():
   return 'Hello World'
hello()   # output => [ 'hello' , 'world' ]
```

Прелесть декораторов заключается в том, что помимо добавления функциональности к выходу метода, они могут даже принимать аргументы для функций и дополнительно модифицировать эти аргументы перед передачей в саму функцию. Внутренняя вложенная функция, то есть функция-"обертка", играет здесь важную роль. Она реализуется для обеспечения инкапсуляции и, таким образом, скрывает себя от глобальной области видимости.

```python
# decorator function to capitalize names
def names_decorator(function):
   def wrapper(arg1, arg2):
       arg1 = arg1.capitalize()
       arg2 = arg2.capitalize()
       string_hello = function(arg1, arg2)
       return string_hello
   return wrapper

@names_decorator
def say_hello(name1, name2):
   return 'Hello ' + name1 + '! Hello ' + name2 + '!'
say_hello('sara', 'ansh')   # output => 'Hello Sara! Hello Ansh!'
```

23. Что такое comprehensions Dict и List?

Python comprehensions, как и декораторы, - это синтаксический сахар, который помогает строить измененные и отфильтрованные списки, словари или множества из заданного списка, словаря или множества. Использование понятий позволяет сэкономить много времени и сэкономить код, который мог бы быть значительно более многословным (содержать больше строк кода). Давайте рассмотрим несколько примеров, в которых понимания могут быть действительно полезны:

Словарные (Dict) comprehension используют фигурные скобки и позволяют создавать новые словари на основе уже существующих.

```python
new_dict = {key: value for key, value in old_dict.items() if value > 2}
```

Выполнение математических операций над всем списком

```python
my_list = [2, 3, 5, 7, 11]
squared_list = [x**2 for x in my_list]    # list comprehension
# output => [4 , 9 , 25 , 49 , 121]
squared_dict = {x:x**2 for x in my_list}    # dict comprehension
# output => {11: 121, 2: 4 , 3: 9 , 5: 25 , 7: 49}
```

Выполнение операций условной фильтрации для всего списка

```python
my_list = [2, 3, 5, 7, 11]
squared_list = [x**2 for x in my_list if x%2 != 0]    # list comprehension
# output => [9 , 25 , 49 , 121]
squared_dict = {x:x**2 for x in my_list if x%2 != 0}    # dict comprehension
# output => {11: 121, 3: 9 , 5: 25 , 7: 49}
```

Объединение нескольких списков в один

Понятия позволяют использовать несколько итераторов и, следовательно, могут быть использованы для объединения нескольких списков в один.

```python
a = [1, 2, 3]
b = [7, 8, 9]
[(x + y) for (x,y) in zip(a,b)]  # parallel iterators
# output => [8, 10, 12]
[(x,y) for x in a for y in b]    # nested iterators
# output => [(1, 7), (1, 8), (1, 9), (2, 7), (2, 8), (2, 9), (3, 7), (3, 8), (3, 9)]
```

Преобразование многомерного массива в одномерный

Аналогичный подход вложенных итераторов (как описано выше) может быть применен для сглаживания многомерного списка или работы с его внутренними элементами.

```python
my_list = [[10,20,30],[40,50,60],[70,80,90]]
flattened = [x for temp in my_list for x in temp]
# output => [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Примечание: Понимание списков имеет тот же эффект, что и метод map в других языках. Они используют математическую нотацию построителя множеств, а не функции `map` и `filter` в Python.