Структуры данных Python

Python - это интерпретируемый язык с динамической типизацией. Что это означает? Если с интерпретируемостью понятно, то динамическая типизация вызывает ряд вопросов. Что это такой за зверь? Если говорят, что язык с динамической типизацией, то это значит, что переменная обретает тип не на этапе объявления, а в момент присваивания ей значения. Динамическая типизация упрощает написание программ, однако затрудняет поиск ошибок на этапе компиляции.

В Python есть изменяемые (mutable) и неизменяемые (immutable) переменные. Если переменная изменяемая, то при присваивании ей нового значения, мы используем все ту же ячейку памяти, меняется лишь значение в ней хранящееся. Если переменная неизменяемая, то в этом случае присваивание переменной нового значения приведет к резервированию новой ячейки памяти с новым значением.

В терминах чтения/записи, изменяемые переменные поддерживают как чтение, так и запись в выбранную ячейку памяти, а неизменяемые - поддерживают только чтение.

Если удобно, неизменяемые переменные как константы, с той лишь разницей, что константной переменной в принципе нельзя присвоить новое значение (имя переменной связывается с ее значением на этапе компиляции и да, в Python констант нет), а неизменяемой можно. Важно другое, сам объект, на который ссылается переменная изменить нельзя.

Откуда вообще появились неизменяемы переменные, были же константные выражения, с ними все было понятно, и вот нате! Неизменяемые переменные!

Давайте вспоминать, константное выражение очень непонятное явление и работа с ним, я имею ввиду на уровне железа, будет разной, в зависимости от того, что за тип у этой переменной.

Если константное выражение это число, то оно жестко прописывается в код программы, память на стеке и куче под него не выделяется. Но что происходит, если константа-это строка или пользовательская структура данных? Здесь единственный способ - это модификация доступа к участку памяти. На уровне компилятора блокировать доступ к таким переменным. Это означает, что любая попытка изменить неизменяемую переменную приведет к ошибке компиляции (в компилируемых языках) или созданию другого объекта - в python.

Другой пример, пусть у нас есть кортеж (будет описан далее)

```
a = (1,2,3,4)

# получить доступ к элементу кортежа я могу
print(a[1])

# но вот попытка заменить часть его
a[1] = 12
# вызовет ошибку
#TypeError: 'tuple' object does not support item assignment

# при этом полностью заменить кортеж мы можем
```

```
a = (2,3,1,4,1,5,6)
```

Простые структуры данных

Здесь я имею ввиду такие типы как:

1. int
2. float
3. str
4. complex
5. bool

По своей природе они являются объектами и по совместительству неизменяемыми. Это значит, что все они имеют дополнитеьльные методы, и запись в них создает новый объект.

Пример:

```
a = 12
print(id(a))
> 4364829264

a+=1
print(id(a))
> 4364829296

a=15
print(id(a))
> 4364829616

b=a
print(id(b), id(a))
> 4364829616 4364829616
```

Как видно, любая попытка изменить значение переменной приводит к созданию нового объекта в памяти с последующим связыванием с переменной. Важно, что присваивание переменной другой переменной - это копирование адреса памяти, где находится значение. В этом можно убедиться посмотрев адреса переменных a и b. Они одинаковы.

Особое место занимает логический тип (bool). Он может принимать всего 2 значения True и False, которые являются неизменяемыми.

```
a = True
b = False
c = True
d = False

print(id(a), id(b), id(c), id(d))
```

```
> 4363916256 4363916984 4363916256 4363916984
```

Как видим, все переменные, имеющие значение True и False имеют одинаковый адрес (соответственно a и c, а также b и d ссылаются на один и тот же объект.

Базовые структуры данных Python

Списки

Одной из базовых структур данных в Python является список list. Он представляет собой структуру, которая может хранить в себе разнородные данные.

Пример:

```
a = list() # инициализация пустого списка
b = [] # то же, упрощенная версия
c = [1,2,'Test',None] # инициализация списка значениями 1, 2, 'Test', None.
```

Доступ к элементу списка осуществляется по его индексу:

Пример:

```
print(c[1]) # нумерация элементов начинается с нуля
> 2
```

Со списком можно делать срезы:

Формат среза имя_списка[a:b] вернет нам подсписок начиная с элемента с индексом a исходного списка до b-1.

```
a = [1,2,3,4,5,6,7,8,9]

b = a[2:6]
print(b)
> [3 4 5 6]
```

Если мы хотим сделать срез начиная с самого первого элемента списка, или начиная с какого-то элемента до конца, в этом случае первый/последний индекс можно опустить

```
print(a[:,4])
> [1 2 3 4]

print(a[3:])
> [4 5 6 7 8 9]
```

Для создания копии списка (shallow copy) можно воспользоваться следующей конструкцией имя_списка[:]

```
b = a
c = a[:]
print(id(a), id(b), id(c))

> 4406527360 4406527360 4424269952
```

Заметьте, если мы присваиваем переменной b имя списка, никакого копирования не происходит, обе переменные ссылаются на один и тот же кусок памяти.

В случае, когда мы присваиваем c = a[:], здесь мы создаем срез нашего исходного списка. В чем здесь особенность? Срез - это тоже список, но он уже другой, размещается по другому адресу и содержит все те же элементы. Если быть точным, создается так называемая shallow copy, копируется сам список, а в него помещаются те же объекты:

```
a = [1,2,3,4,5]
b = a
c = a[:]

print(id(a), id(b), id(c))
> 4406527360 4406527360 4424269952 # несмотря на то, что списки a и c - разные


for i in range(len(a)):
    print(id(a[i]), id(b[I]), id(c[i]))

> 4364828912 4364828912 4364828912 # содержат они те же элементы, что и исходный
> 4364828944 4364828944 4364828944 # я имею ввиду объекты те же
> 4364828976 4364828976 4364828976 #
> 4364829008 4364829008 4364829008 #
> 4364829040 4364829040 4364829040 #
```

Что означает, что объекты те же?

Рассмотрим пример:

```
a = [[1,2],[3,4], [5,6]] # пусть у нас будет список списков
b = a
c = a[:]

# так как a и b ссылаются на один и тот же участок памяти, то
b[0][0]=34
print(a)
> [[34, 2], [3, 4], [5, 6]] # изменилось и значение в списке a, так и должно быть, a и b - это одна и та же область памяти

print(c)
> [[34, 2], [3, 4], [5, 6]] # и c тоже изменился, почему? адрес же списка c другой? Другой-то он другой, но содержит же ровно те же объекты! Списки из двух элементов. Поэтому он тоже изменился.
```

Когда мы забываем об этой особенности срезов и копирования объектов, возникают

трудноуловимые ошибки.

Добавление элементов в список Существуют две конструкции добавления элементов в список:

1. Использование метода append (элемент добавляется в исходный список)
2. Использование выражения +=[] (элемент добавляется в копию исходного списка)

```
a = list()
b = []

a.append(33)
a.append('123')

b=b+[33]
b=b+['123']

print(a, b)
> [33, '123'] [33, '123']
```

Для списка определены методы `append()`, `clear()`, `copy()`, `count()`, `extend()`, `index()`, `insert()`, `pop()`, `remove()`, `reverse()`, `sort()`. При этом методы, которые направлены на изменение списка меняют сам объект, а не создают его копию.

Кортежи

Итак, кортежи по своей сути, тоже коллекция, за за исключением операций изменения. Здесь я имею ввиду изменения самого кортежа (операции добавления или удаления элементов, сортировка, удаление элементов, разворот и прочие). Эти операции для него попросту не определены. Во всех остальных аспектах он ведет вея ровно так же как и список. Если быть совсем точным, то остается лишь две операции, допустимые над кортежем:

- вычисление длины кортежа
- поиск первого вхождения элемента в кортеж

Создается кортеж по аналогии со списком, используя ключевое слово `tuple()` или конструкцию `()`

Пример:

```
a = tuple()       # создается пустой кортеж
b = ()            # создается пустой кортеж
c = (1,2,3,4,5)   # создается кортеж с элементами 1,2,3,4,5
```

В кортежах разрешен доступ к элементу по его индексу, нумерация элементов начинается с нуля:

```
a = (1,2,3,4,5)
print( a[2] )
> 3
```

Выше мы рассмотрели операции, которые предоставляет кортеж и которые не изменяют его.

В стандартной библиотеке python над кортежем все-таки определены операции добавления. Отличие от списка состоит в том, что при добавлении в кортеж создается его новая копия, в которую добавлен новый элемент.

Пример:

```
a = [1,2,3,4] # список
b = (1,2,3,4) # кортеж

print(id(a), id(b)) # выводим адрес в памяти для объектов
> 4556940352 4556316304

a = a.append(1) # добавляем в конец списка элемент 1
b = b + (1,)    # добавляем в конец кортежа элемент 1

print(id(a), id(b)) # выводим адрес в памяти для объектов
> 4556940352 4556311345
```

Поскольку кортеж это тот же список без операций модификации, для списка операция `a = a + [1,]` тоже создаст новый список.

Словари

Словарь или хеш-таблица (будем для однородности называть его словарем) - это такая структура данных, которая хранит соответствие ключ→значение. Если по-другому, это массив, индексами которого могут быть произвольные элементы, не только числа. Словарь динамическая структура данных.

Множества

Динамические структуры данных

Так, давайте разберемся, о чем же здесь пойдет речь. Что за звери такие, эти динамические структуры данных? Почему их выносят в отдельный класс?

Определение: динамическая структура данных - это любая структура даных, размер которой не является фиксированным, то есть, может изменяться во время выполнения программы.

Односвязный список

Двусвязный список

Очередь

Стек