Введение в аннотации типов Python

Python — язык с динамической типизацией и позволяет нам довольно вольно оперировать переменными разных типов. Однако при написании кода мы так или иначе предполагаем переменные каких типов будут использоваться (это может быть вызвано ограничением алгоритма или бизнес логики). И для корректной работы программы нам важно как можно раньше найти ошибки, связанные с передачей данных неверного типа.

Сохраняя идею динамической утиной типизации в современных версиях Python (3.6+) поддерживает аннотации типов переменных, полей класса, аргументов и возвращаемых значений функций:

Аннотации типов просто считываются интерпретатором Python и никак более не обрабатываются, но доступны для использования из стороннего кода и в первую очередь рассчитаны для использования статическими анализаторами.

Аннотации типов поддерживаются многими IDE для Python, которые выделяют некорректный код или выдают подсказки в процессе набора текста.

Так же аннотации типов обрабатываются и консольными линтерами. pylint: mypy:

**Основы**

В отличие от старых версий Python, аннотации типов пишутся не в комментариях или docstring, а непосредственно в коде. С одной стороны, это ломает обратную совместимость, с другой — явно означает что это часть кода и может обрабатываться соответственно

В простейшем случае аннотация содержит непосредственно ожидаемый тип. Более сложные кейсы будут рассмотрены ниже. Если в качестве аннотации указан базовый класс, допустимо передача экземпляров его наследников в качестве значений. Однако использовать можно только те возможности, что реализованы в базовом классе.

Аннотации для переменных пишут через двоеточие после идентификатора. После этого может идти инициализация значения. Например,

price: int = 5

title: str

**def** **indent\_right**(s: str, width: int) -> str:

**return** " " \* (max(0, width - len(s))) + s

Параметры функции аннотируются так же как переменные, а возвращаемое значение указывается после стрелки -> и до завершающего двоеточия.

Для полей класса аннотации должны быть указаны явно при определении класса. Однако анализаторы могут выводить автоматически их на основе \_\_init\_\_ метода, но в этом случае они не будут доступны во время выполнения программы. Подробнее про работу с аннотациями в рантайме во второй части статьи

**class** **Book**:

title: str

author: str

**def** **\_\_init\_\_**(self, title: str, author: str) -> None:

self.title = title

self.author = author

b: Book = Book(title='Fahrenheit 451', author='Bradbury')

**Встроенные типы**

Хоть вы и можете использовать стандартные типы в качестве аннотаций, много полезного сокрыто в модуле typing.

**Optional**

Если вы пометите переменную типом int и попытаетесь присвоить ей None, будет ошибка:

Для таких случаев предусмотрена в модуле typing аннотация Optional с указанием конкретного типа. Обратите внимание, тип опциональной переменной указывается в **квадратных** скобках

**from** typing **import** Optional

amount: int

amount = None # Incompatible types in assignment (expression has type "None", variable has type "int")

price: Optional[int]

price = None

**Any**

Иногда вы не хотите ограничивать возможные типы переменной. Например, если это действительно не важно, или если вы планируете сделать обработку разных типов самостоятельно. В этом случае, можно использовать аннотацию Any.

unknown\_item: Any = 1

print(unknown\_item)

print(unknown\_item.startswith("hello"))

print(unknown\_item // 0)

**Union**

Для случаев, когда необходимо допустить использование не любых типов, а только некоторых, можно использовать аннотацию typing. Union с указанием списка типов в квадратных скобках.

**def** **hundreds**(x: Union[int, float]) -> int:

**return** (int(x) // 100) % 10

hundreds(100.0)

hundreds(100)

hundreds("100") # Argument 1 to "hundreds" has incompatible type "str"; expected "Union[int, float]"

**Коллекции**

Механизм аннотаций типов поддерживает механизм дженериков, которые позволяют специфицировать для контейнеров типы элементов, хранящихся в них.

**Списки**

Для того, чтобы указать, что переменная содержит список можно использовать тип list в качестве аннотации. Однако если хочется конкретизировать, какие элементы содержит список, он такая аннотация уже не подойдёт. Для этого есть typing.List. Аналогично тому, как мы указывали тип опциональной переменной, мы указываем тип элементов списка в квадратных скобках.

titles: List[str] = ["hello", "world"]

titles.append(100500) # Argument 1 to "append" of "list" has incompatible type "int"; expected "str"

titles = ["hello", 1] # List item 1 has incompatible type "int"; expected "str"

items: List = ["hello", 1]

Предполагается, что список содержит неопределенное количество однотипных элементов. Но при этом нет ограничений на аннотацию элемента: можно использовать Any, Optional, List и другие. Если тип элемента не указан, предполагается, что это Any.

Кроме списка аналогичные аннотации есть для множеств: typing.Set и typing.FrozenSet.

**Кортежи**

Кортежи в отличие от списков часто используются для разнотипных элементов. Синтаксис похож с одним отличием: в квадратных скобках указывается тип каждого элемента кортежа по отдельности.

Если же планируется использовать кортеж аналогично списку: хранить неизвестное количество однотипных элементов, можно воспользоваться многоточием (...).

Аннотация Tuple без указания типов элементов работает аналогично Tuple[Any, ...]

price\_container: Tuple[int] = (1,)

price\_container = ("hello") # Incompatible types in assignment (expression has type "str", variable has type "Tuple[int]")

price\_container = (1, 2) # Incompatible types in assignment (expression has type "Tuple[int, int]", variable has type "Tuple[int]")

price\_with\_title: Tuple[int, str] = (1, "hello")

prices: Tuple[int, ...] = (1, 2)

prices = (1, )

prices = (1, "str") # Incompatible types in assignment (expression has type "Tuple[int, str]", variable has type "Tuple[int, ...]")

something: Tuple = (1, 2, "hello")

**Словари**

Для словарей используется typing.Dict. Отдельно аннотируется тип ключа и тип значений:

book\_authors: Dict[str, str] = {"Fahrenheit 451": "Bradbury"}

book\_authors["1984"] = 0 # Incompatible types in assignment (expression has type "int", target has type "str")

book\_authors[1984] = "Orwell"

Аналогично используются typing.DefaultDict и typing.OrderedDict

**Результат выполнения функции**

Для указания типа результата функции можно использовать любую аннотацию. Но есть несколько особенных случаев.

Если функция ничего не возвращает (например, как print), её результат всегда равен None. Для аннотации так же используем None.

Корректными вариантами завершения такой функции будут: явный возврат None, возврат без указания значения и завершение без вызова return.

**def** **nothing**(a: int) -> None:

**if** a == 1:

**return**

**elif** a == 2:

**return** None

**elif** a == 3:

**return** "" # No return value expected

**else**:

**pass**

Если же функция **никогда** не возвращает управление (например, как sys.exit), следует использовать аннотацию NoReturn:

**def** **forever**() -> NoReturn:

**while** True:

**pass**

Если это генераторная функция, то есть её тело содержит оператор yield, для возвращаемого можно воспользоватьтся аннотацией Iterable[T], либо [Generator[YT, ST, RT]](https://docs.python.org/3/library/typing.html" \l "typing.Generator):

**def** **generate\_two**() -> Iterable[int]:

**yield** 1

**yield** "2" # Incompatible types in "yield" (actual type "str", expected type "int")

### Предварительное объявление

Обычно вы не можете использовать тип до того, как он создан. Например, следующий код даже не запустится:

**class** **LinkedList**:

data: Any

next: LinkedList # NameError: name 'LinkedList' is not defined

Чтобы это исправить, допустимо использовать строковый литарал. В этом случае аннотации будут вычислены отложенно.

**class** **LinkedList**:

data: Any

next: 'LinkedList'

Так же вы можете обращаться к классам из других модулей (конечно, если модуль импортирован): some\_variable: 'somemodule.SomeClass'

### Функции и вызываемые объекты

Для ситуаций, когда необходимо передать функцию или другой вызываем объект (например, в качестве callback) нужно использовать аннотацию Callable[[ArgType1, ArgType2,...], ReturnType] Например,

**def** **help**() -> None:

print("This is help string")

**def** **render\_hundreds**(num: int) -> str:

**return** str(num // 100)

**def** **app**(helper: Callable[[], None], renderer: Callable[[int], str]):

helper()

num = 12345

print(renderer(num))

app(help, render\_hundreds)

app(help, help) # error: Argument 2 to "app" has incompatible type "Callable[[], None]"; expected "Callable[[int], str]"

Допустимо указать только возвращаемый тип функции без указания её параметров. В этом случае используется многоточие: Callable[..., ReturnType]. Обратите внимание, что квадратные скобки вокруг многоточия отсутствуют.

На текущий момент невозможно описать сигнатуру функции с переменным числом параметров определенного типа или указать именованные аргументы.

### Generic-типы

Иногда необходимо сохранить информацию о типе, при этом не фиксируя его жестко. Например, если вы пишете контейнер, который хранит однотипные данные. Или функцию, которая возвращает данные того же типа, что и один из аргументов.

Такие типы как List или Callable, которые, мы видели раньше как раз используют механизм дженериков. Но кроме стандартных типов, вы можете создать свои дженерик-типы.

Для этого надо, во-первых, завести TypeVar переменную, которая будет атрибутом дженерика, и, во-вторых, непосредственно объявить generic-тип:

T = TypeVar("T")

**class** **LinkedList**(Generic[T]):

data: T

next: "LinkedList[T]"

**def** **\_\_init\_\_**(self, data: T):

self.data = data

head\_int: LinkedList[int] = LinkedList(1)

head\_int.next = LinkedList(2)

head\_int.next = 2

head\_int.data += 1

head\_int.data.replace("0", "1") # error: "int" has no attribute "replace"

head\_str: LinkedList[str] = LinkedList("1")

head\_str.data.replace("0", "1")

head\_str = LinkedList[str](1)

### Cast

Иногда анализатор статический анализатор не может корректно определить тип переменной, в этом случае можно использовать функцию cast. Её единственная задача — показать анализатору, что выражение имеет определённый тип. Также это может быть полезно для декораторов:

**from** typing **import** List, cast

**def** **find\_first\_str**(a: List[object]) -> str:

index = next(i **for** i, x **in** enumerate(a) **if** isinstance(x, str))

**return** cast(str, a[index])

MyCallable = TypeVar("MyCallable", bound=Callable)

**def** **logged**(func: MyCallable) -> MyCallable:

@wraps(func)

**def** **wrapper**(\*args, \*\*kwargs):

print(func.\_\_name\_\_, args, kwargs)

**return** func(\*args, \*\*kwargs)

**return** cast(MyCallable, wrapper)

@logged

**def** **mysum**(a: int, b: int) -> int:

**return** a + b

mysum(a=1) # error: Missing positional argument "b" in call to "mysum"

**Линейный поиск в Python**

Поиск – это метод определения того, присутствует ли конкретный элемент в данном списке.

Есть два типа поиска: линейный поиск, бинарный поиск.

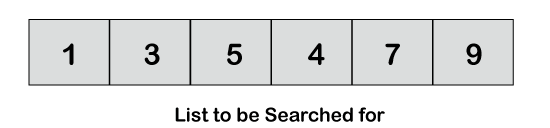
Оба метода широко используются для поиска элемента в данном списке.

**Что такое линейный поиск?**

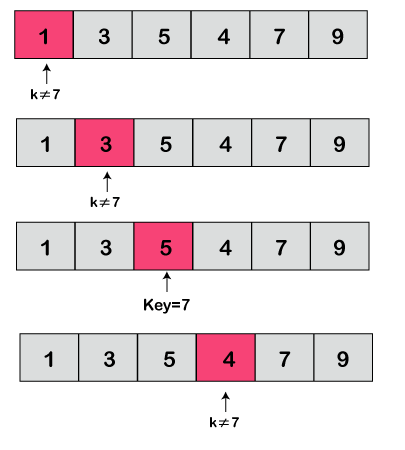
Линейный поиск – это метод поиска элементов в списке. Его еще называют последовательным поиском. Это простейший алгоритм поиска, поскольку он ищет желаемый элемент последовательно. Он сравнивает каждый элемент со значением, которое мы ищем. Если оба совпадают, элемент найден, и алгоритм возвращает позицию индекса ключа.

**Концепция линейного поиска**

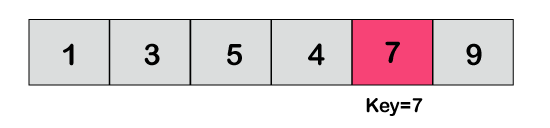
Давайте разберемся в следующих шагах, чтобы найти элемент key = 7 в данном списке.

Шаг – 1: Начните поиск с первого элемента и проверьте ключ = 7 с каждым элементом списка x.

Шаг – 2: Если элемент найден, вернуть индексную позицию ключа.



Шаг – 3: Если элемент не найден, возвращаемого элемента нет.



Давайте разберемся в следующей реализации алгоритма линейного поиска на Python.

Сложность линейного поиска

Временная сложность алгоритма линейного поиска:

Базовый вариант – O (1) Средний случай – O (n) Худший случай -O (n)

Алгоритм линейного поиска подходит для небольшого списка (<100), потому что он проверяет каждый элемент, чтобы получить желаемое число. Предположим, что имеется список из 10 000 элементов и желаемый элемент доступен в последней позиции, это займет много времени при сравнении с каждым элементом списка.

**Подвиг**

Можно использовать только 1 цикл!!!

Напишите программу, которая находит в массиве элемент, самый близкий по величине к данному числу.

Входные данные:

- задается натуральное число N, не превосходящее 1000 – размер массива.

- в списке содержатся N чисел (целые числа, не превосходящие по модулю 1000).

- вводится одно целое число X, не превосходящее по модулю 1000.

Вывести значение элемента массива, ближайшее к X. Если таких чисел несколько, выведите любое из них.

**Примеры**

входные данные

5

1 2 3 4 5

6

выходные данные

5

входные данные

5

5 4 3 2 1

3

выходные данные

3

**Бинарный поиск: зачем нужен и как реализовать**

Бинарный поиск – это алгоритм поиска элемента в отсортированном массиве данных. Этот метод является довольно популярным в программировании и может быть реализован на разных языках: от С до Python.

Несмотря на свою простоту, у бинарного поиска есть ряд сложностей в реализации. Даже опытные программисты часто ошибаются при работе с данным алгоритмом.

Бинарным (или двоичным) называют поиск элемента упорядоченного множества через многократное деление этого множества пополам. Искомый элемент всегда будет оказываться в одной из двух частей. Поиск прекращается, когда обнаруживается совпадение граничного элемента между двумя разделенными блоками с заданным, или когда заданный элемент не обнаруживается вовсе.

Реализация этого метода возможна только применимо к отсортированным множествам. Последовательно разбивая такой массив данных на две части, алгоритм каждый раз ищет заданный элемент только в одной половине.

В целом метод бинарного поиска можно описать следующим образом. Сначала в возрастающем или убывающем множестве определяется среднее значение, после чего оно сравнивается с искомым. При совпадении заданного и центрального элемента поиск прекращается — элемент считается найденным. В случае несовпадения значений создается новый массив значений соответственно слева и справа от среднего, и процедура повторяется уже на данном массиве.

**Принцип работы бинарного поиска на примере**

Принцип достаточно прост. Множество данных предварительно сортируется (чаще всего по возрастанию). Затем для поиска конкретных элементов выполняется следующая последовательность действий:

* Вычисляется среднее значение массива.
* Значение полученного элемента сравнивается с искомым (ключом). Если оно меньше, дальнейший поиск для возрастающего массива выполняется слева от центрального элемента. В противном случае ключ ищется справа.
* В случае совпадения среднего значения с искомым поиск прекращается. Пользователю возвращается индекс совпавшего элемента.
* Дальнейшие итерации первых двух шагов повторяются вплоть до нахождения ключа.
* Если в результате очередного деления остался лишь один элемент, и он не совпадает с искомым, пользователю возвращается значение -1.

Имеется массив данных: 1, 3, 6, 7, 13, 15, 16, 19, 24, 28, 29.

Мы привели простой пример, содержащий небольшое число шагов до обнаружения искомого элемента. Итого, за 4 итерации удалось обнаружить ключ в последовательности из 11 чисел.

**Бинарный поиск работает по принципу «разделяй и властвуй».**

Он быстрее, чем линейный поиск, но требует, чтобы массив был отсортирован перед выполнением алгоритма. Предполагая, что мы ищем значение val в отсортированном массиве, алгоритм сравнивает val со значением среднего элемента массива, который мы будем называть mid. Если mid — это тот элемент, который мы ищем (в лучшем случае), мы возвращаем его индекс.

Если нет, мы определяем, в какой половине массива мы будем искать val дальше, основываясь на том, меньше или больше значение val значения mid, и отбрасываем вторую половину массива.

Затем мы рекурсивно или итеративно выполняем те же шаги, выбирая новое значение для mid, сравнивая его с val и отбрасывая половину массива на каждой итерации алгоритма.

Алгоритм бинарного поиска можно написать как рекурсивно, так и итеративно.

Мы можем выбрать только одно действие на каждой итерации. Также на каждой итерации наш массив делится на две части. Из-за этого временная сложность двоичного поиска равна O(log n).

Одним из недостатков бинарного поиска является то, что если в массиве имеется несколько вхождений элемента, он возвращает индекс не первого элемента, а ближайшего к середине:

>>> print(BinarySearch([4,4,4,4,4], 4))

После выполнения этого фрагмента кода будет возвращен индекс среднего элемента:

Для сравнения: выполнение линейного поиска по тому же массиву вернет индекс первого элемента:З

Сложность бинарного поиска, несмотря на сравнительную простоту его алгоритма, заключается в самой практической реализации. Здесь полезно привести слова упомянутого выше Йона Бентли, которые он адресовал своим студентам:

Многие программисты убеждены, что лишь обладая полноценным описанием алгоритма двоичного поиска, они способны легко написать для этого алгоритма программу. Но это является заблуждением. Попробуйте самостоятельно реализовать поиск в виде кода и убедитесь, что эта задача — не из легких.

Итак, алгоритм бинарного поиска по своей сути простой, но при этом правильно реализовать его достаточно сложно. В книге Дональда Кнута «Sorting and Searching» говорится о первой публикации идеи двоичного поиска. Впервые принцип был опубликован в 1946 году, но лишь спустя 12 лет этот принцип грамотно реализовали в виде кода программы.

По заверению Йона Бентли подобную задачу он задавал не только своим студентам, но и профессиональным разработчикам. Как уже упоминалось, лишь десятая часть начинающих и опытных программистов с первого раза справлялась с заданием. Курьез заключается в том, что собственный код Бентли содержит ошибки. И по сей день подобные ошибки обнаружить весьма непросто.

Так, почти все выпущенные на 2006 год версии JDK содержали некорректно разработанные алгоритмы двоичного поиска. Скорее всего, спустя полтора десятилетия ситуация кардинально не изменилась. Столь печальный вывод должен служить примером недопустимости слепого копирования исходного кода. Практика показывает, что ошибки могут подстерегать разработчика даже в простейших по структуре алгоритмах.

Рассмотренные примеры показывают принцип работы бинарного дерева поиска. Алгоритм реализуется практически во всех известных языках программирования. Он эффективен в большинстве случаев при грамотной реализации, но для его нормального функционирования необходимо заранее отсортировать массив нужным образом.