MLStack.Cafe - Kill Your Data Science & ML Interview

Q1: What are the *built-in types* available In Python? ☆

Topics: Python

Answer:

Common *immutable* type:

1. numbers: `int()`, `float()`, `complex()`
2. immutable sequences: `str()`, `tuple()`, `frozenset()`, `bytes()`

Common *mutable* type (almost everything else):

1. mutable sequences: `list()`, `bytearray()`
2. set type: `set()`
3. mapping type: `dict()`
4. classes, class instances
5. etc.

You have to understand that Python represents all its data as objects. Some of these objects like lists and dictionaries are mutable, meaning you can change their content without changing their identity. Other objects like integers, floats, strings and tuples are objects that can not be changed.

Q2: Name some *characteristics* of Python? ☆

Topics: Python

Answer:

Here are a few key points:

- Python is an **interpreted language**. That means that, unlike languages like C and its variants, Python does not need to be compiled before it is run. Other interpreted languages include *PHP* and *Ruby*.
- Python is **dynamically typed**, this means that you don't need to state the types of variables when you declare them or anything like that. You can do things like `x=111` and then `x="I'm a string"` without error
- Python is well suited to **object orientated programming** in that it allows the definition of classes along with composition and inheritance. Python does not have access specifiers (like C++'s `public`, `private`), the justification for this point is given as "we are all adults here"
- In Python, **functions are first-class objects**. This means that they can be assigned to variables, returned from other functions and passed into functions. Classes are also first class objects
- Writing Python code is quick but running it is **often slower than compiled languages**. Fortunately, Python allows the inclusion of C based extensions so bottlenecks can be optimised away and often are. The `numpy` package is a good example of this, it's really quite quick because a lot of the number crunching it does isn't actually done by Python

Q3: How do I *modify* a string? ☆

Topics: Python

Answer:

You can't because strings are *immutable*. In most situations, you should simply construct a new string from the various parts you want to assemble it from. Work with them as lists; turn them into strings only when needed.

```
>>> s = list("Hello zorld")
>>> s
['H', 'e', 'l', 'l', 'o', ' ', 'z', 'o', 'r', 'l', 'd']
>>> s[6] = 'W'
>>> s
['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
>>> "".join(s)
'Hello World'
```

Q4: Name some *benefits* of Python ☆☆

Topics: Python

Answer:

- Python is a **dynamic-typed** language. It means that you don't need to mention the data type of variables during their declaration.
- Python supports **object-orientated programming** as you can define classes along with the composition and inheritance.
- **Functions** in Python are like **first-class objects**. It suggests you can assign them to variables, return from other methods and pass them as arguments.
- Developing using Python is quick but running it is often slower than compiled languages.
- Python has several usages like web-based applications, test automation, data modeling, big data analytics, and much more.

Q5: What is *Lambda Functions* in Python? ☆☆

Topics: Python

Answer:

A **Lambda Function** is a small anonymous function. A lambda function can take *any* number of arguments, but can *only* have *one* expression.

Consider:

```
x = lambda a : a + 10
print(x(5)) # Output: 15
```

Q6: When to use a `tuple` vs `list` vs `dictionary` in Python? ☆☆

Topics: Python

Answer:

- Use a `tuple` to store a sequence of items that *will not change*.
- Use a `list` to store a sequence of items that *may change*.
- Use a `dictionary` when you want to associate *pairs* of two items.

Q7: What is *Negative Index* in Python? ☆☆

Topics: Python

Answer:

Negative numbers mean that you count from the right instead of the left. So, `list[-1]` refers to the last element, `list[-2]` is the second-last, and so on.

Q8: What are *local variables* and *global variables* in Python? ☆☆

Topics: Python

Answer:

- **Global Variables**: Variables declared *outside* a function or in global space are called global variables. These variables can be accessed by any function in the program.
- **Local Variables**: Any variable declared *inside* a function is known as a local variable. This variable is present in the local space and not in the global space.

Q9: What are the rules for *local* and *global* variables in Python? ☆☆

Topics: Python

Answer:

While in many or most other programming languages variables are treated as global if not declared otherwise, Python deals with variables the other way around. They are local, if not otherwise declared.

- In Python, variables that are only referenced inside a function are implicitly *global*.
- If a variable is assigned a value anywhere within the function's body, it's assumed to be a *local* unless explicitly declared as global.

Requiring global for assigned variables provides a bar against unintended side-effects.

Q10: Does Python have a *switch-case* statement? ☆☆

Topics: Python

Answer:

In Python befor 3.10, we **do not hav**e a switch-case statement. Here, you may write a switch function to use. Else, you may use a set of if-elif-else statements. To implement a function for this, we may use a dictionary.

```
def switch_demo(argument):
    switcher = {
        1: "January",
        2: "February",
        3: "March",
```

```
        4: "April",
        5: "May",
        6: "June",
        7: "July",
        8: "August",
        9: "September",
        10: "October",
        11: "November",
        12: "December"
    }
    print switcher.get(argument, "Invalid month")
```

Python 3.10 (2021) introduced the `match` - `case` statement which provides a first-class implementation of a "switch" for Python. For example:

For example:

```
def f(x):
    match x:
        case 'a':
            return 1
        case 'b':
            return 2
```

The `match` - `case` statement is considerably more powerful than this simple example.

Q11: How the *string* does get converted to a *number*? ☆☆

Topics: Python

Answer:

- To convert the string into a number the built-in functions are used like `int()` constructor. It is a data type that is used like `int ('1') == 1`.
- `float()` is also used to show the number in the format as `float('1') = 1`.
- The number by default are interpreted as decimal and if it is represented by `int('0x1')` then it gives an error as `ValueError`. In this the `int(string,base)` function takes the parameter to convert string to number in this the process will be like `int('0x1',16) == 16`. If the base parameter is defined as 0 then it is indicated by an octal and 0x indicates it as hexadecimal number.
- There is function `eval()` that can be used to convert string into number but it is a bit slower and present many security risks

Q12: What are descriptors? ☆☆

Topics: Python

Answer:

Descriptors were introduced to Python way back in version 2.2. They provide the developer with the ability to add managed attributes to objects. The methods needed to create a descriptor are `__get__`, `__set__` and `__delete__`. If you define any of these methods, then you have created a descriptor.

Descriptors power a lot of the magic of Python's internals. They are what make properties, methods and even the super function work. They are also used to implement the new style classes that were also introduced in Python 2.2.

Q13: Explain what is Linear (Sequential) Search and when may we use one? ☆☆

Topics: Searching Python

Answer:

Linear (sequential) search goes through all possible elements in some array and compare each one with the desired element. It may take up to $O(n)$ operations, where N is the size of an array and is widely considered to be horribly slow. In linear search when you perform one operation you reduce the size of the problem *by one* (when you do one operation in binary search you reduce the size of the problem *by half*). Despite it, it can still be used when:

- You need to perform this search only once,
- You are *forbidden* to rearrange the elements and you do not have any extra memory,
- The array is tiny, such as ten elements or less, or the performance is not an issue at all,
- Even though in theory other search algorithms may be faster than linear search (for instance binary search), in practice even on medium-sized arrays (around 100 items or less) it might be infeasible to use anything else. On larger arrays, it only makes sense to use other, faster search methods if the data is large enough, because the initial time to prepare (sort) the data is comparable to many linear searches,
- When the list items are arranged in order of *decreasing probability*, and these probabilities are geometrically distributed, the cost of linear search is only $O(1)$
- You have no idea what you are searching.

When you ask MySQL something like `SELECT x FROM y WHERE z = t`, and `z` is a column *without* an index, linear search is performed with all the consequences of it. This is why adding an index to *searchable* columns is important.

Complexity Analysis:

Time Complexity: O(n) **Space Complexity**: O(n)

- A linear search runs in at worst *linear time* and makes at most `n` comparisons, where `n` is the length of the list. If each element is *equally likely* to be searched, then linear search has an average case of `(n+1)/2` comparisons, but the average case can be affected if the search probabilities for each element vary.
- When the list items are arranged in order of *decreasing probability*, and these probabilities are geometrically distributed, the cost of linear search is only $O(1)$

Implementation:

JS

```
function linearSearch(array, toFind){
  for(let i = 0; i < array.length; i++){
    if(array[i] === toFind) return i;
  }
  return -1;
}
```

PY

```
# can be simply done using 'in' operator
if x in arr:
   print arr.index(x)
```

```python
# If you want to implement Linear Search in Python
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i

    return -1
```

Q14: Explain what is Binary Search ☆☆

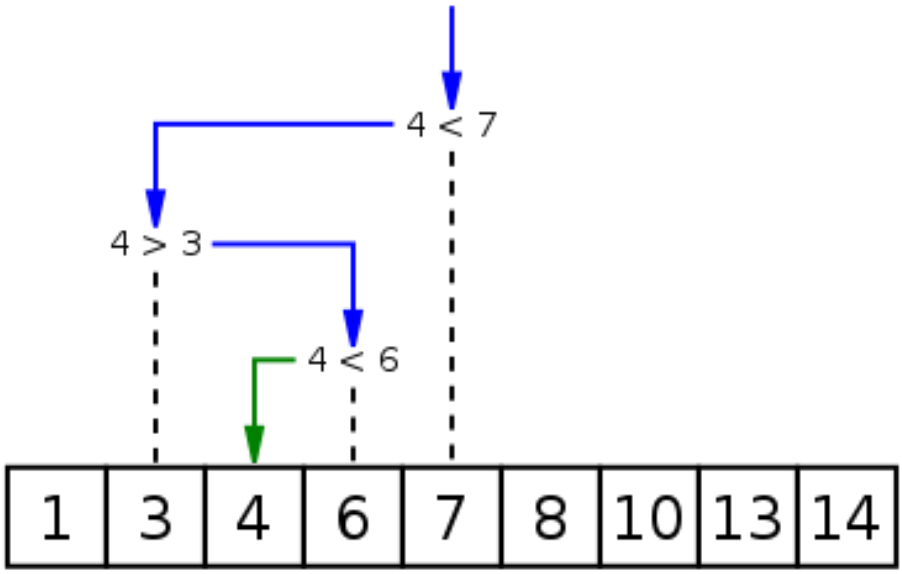
Topics: Searching Python

Answer:

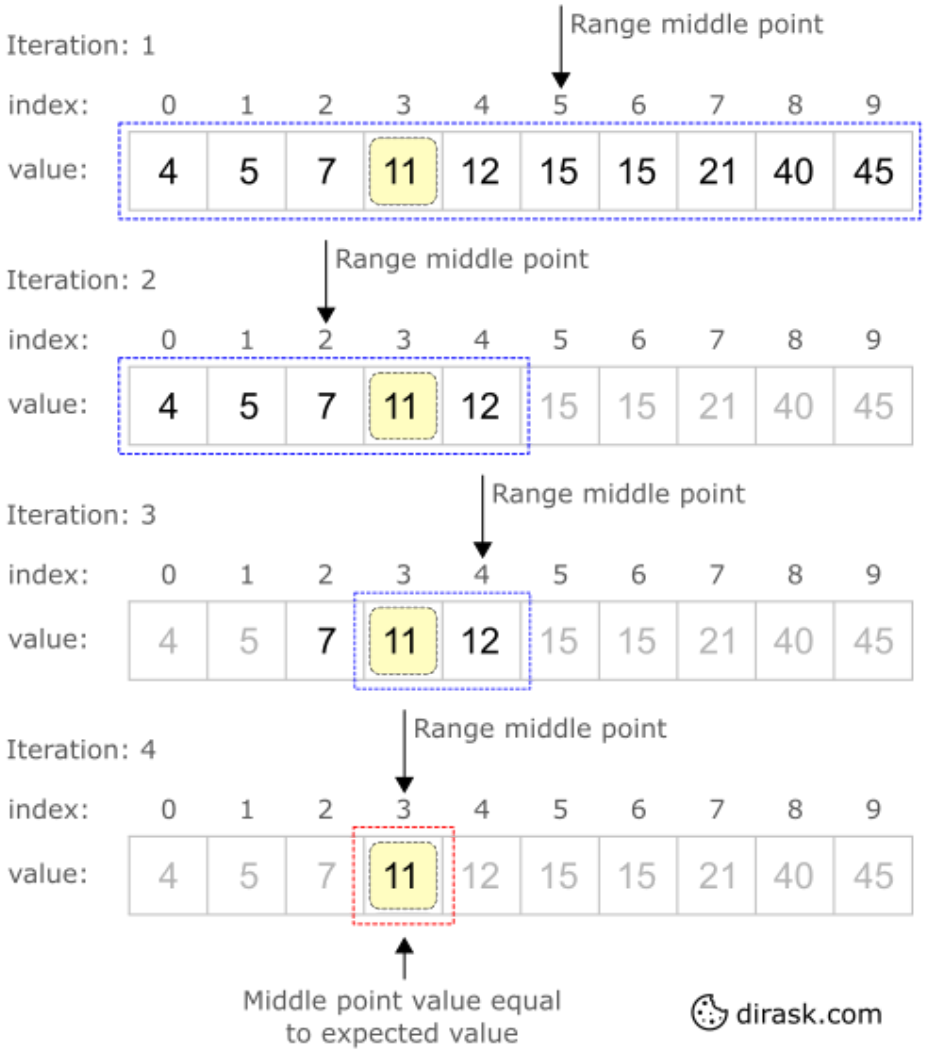
When the list is **sorted** we can use the **binary search** (also known as half-interval search, logarithmic search, or binary chop) technique to find items on the list. Here's a step-by-step description of using binary search:

1. Let `min = 1` and `max = n`.
2. Guess the average of `max` and `min` **rounded down** so that it is an integer.
3. If you guessed the number, stop. You found it!
4. If the guess was too low, set *min* to be one larger than the guess.
5. If the guess was too high, set *max* to be one smaller than the guess.
6. Go back to step two.

In this example we looking for array item with value `4`:



When you do one operation in binary search we reduce the size of the problem **by half** (look at the picture below how do we reduce the size of the problem area) hence the complexity of binary search is $O(log\ n)$. The binary search algorithm can be written either *recursively* or *iteratively*.



Complexity Analysis:

Time Complexity: O(log n) **Space Complexity**: O(log n)

Implementation:

JS

```
var binarySearch = function(array, value) {
    var guess,
        min = 0,
        max = array.length - 1;

    while(min <= max){
        guess = Math.floor((min + max) /2);
    if(array[guess] === value)
        return guess;
    else if(array[guess] < value)
        min = guess + 1;
    else
        max = guess - 1;
     }

     return -1;
}
```

Java

```java
// binary search example in Java
/* here Arr is an of integer type, n is size of array
   and target is element to be found */

int binarySearch(int Arr[], int n, int target) {

    //set stating and ending index
    int start = 0, ending = n-1;

    while(start <= ending) {
        // take mid of the list
        int mid = (start + end) / 2;

        // we found a match
        if(Arr[mid] == target) {
            return mid;
        }
        // go on right side
        else if(Arr[mid] < target) {
            start = mid + 1;
        }
        // go on left side
        else {
            end = mid - 1;
        }
    }
    // element is not present in list
    return -1;
}
```

PY

```python
def BinarySearch(lys, val):
    first = 0
    last = len(lys)-1
    index = -1
    while (first <= last) and (index == -1):
        mid = (first+last)//2
        if lys[mid] == val:
            index = mid
        else:
            if val<lys[mid]:
                last = mid -1
            else:
                first = mid +1
    return index
```