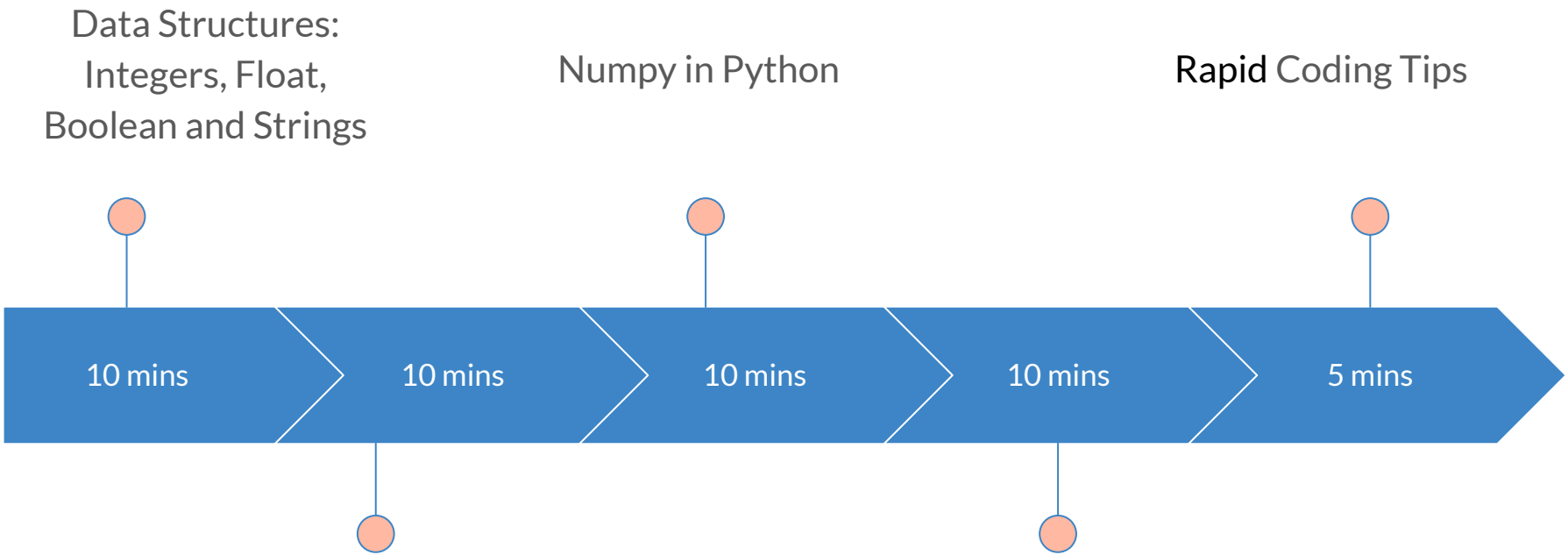
IEEE Student Branch
UNIVERSITY OF MELBOURNE

Data Structures and Algorithms in Python

Sattar Seifollahi, RMIT
sattar.seifollahi@rmit.edu.au
September 17, 2020


Data Structures: Integers, Float, Boolean and Strings
Numpy in Python
Rapid Coding Tips
10 mins
10 mins
10 mins
10 mins
5 mins
Data Structures: Arrays, Lists, Tuples
Algorithms and Functions

Data Structures

Data Structures

Data structures are a way of organizing and storing data so that they can be accessed and worked with efficiently. They define the relationship between the data, and the operations that can be performed on the data. Data structures can be divided into two categories:

1. Primitive data structures; e.g. x, y = 2, 3 or is_correct = True
2. Non-primitive data structures; e.g. list_num = [1,5,3,5]

The former are the simplest forms of representing data, whereas the latter are more advanced and contain the primitive data structures within more complex data structures for special purposes.

Note

You do not have to explicitly state the type of the variable or your data as Python is a dynamically typed language.

Dynamically typed languages are the languages where the type of data an object can store is mutable.

Primitive Data Structures

- Integer
- Float
- String
- Boolean

Integers and Floats

Integers: You can use an integer represent numeric data, and more specifically, whole numbers from negative infinity to infinity, like 4, 5, or -1.

Float: "Float" stands for 'floating point number'. You can use it for rational numbers, usually ending with a decimal figure, such as 1.11 or 3.14.

Integers and Floats

Basic Operations

```
x = 10
y = 3.5
>>> print(type(x),type(y)) # Type of data
>>> x*y      # Multiplication
>>> x**y   # Power
>>> x/y  → 2.85     # Quotient
>>> x//y → 2.0    # Floor division
>>> x%y  → 3.0  # Remainder/modulo
>>> abs(x)   # Absolute value
>>> isinstance(x, int) # is integer?
>>> isinstance(x, float)  # is float?
>>> int(3.4)     # take the integer part
>>> complex("1+2j") or complex(1,2) # complex
```

String

Strings are collections of alphabets, words or other characters. A string can contain letters, numbers, and symbols.

In Python, you can create strings by enclosing a sequence of characters within a pair of single or double quotes.

name = “Ryan” age = “19” food = “cheese” code = “ab234”

Strings

Basic Operations

x, y = 'Apple Cake', 'Cookie'

>>> **x+" & "+y** → 'Apple Cake & Cookie'

>>> **y* 2** → ''CookieCookie'' # repeat

>>> **x[2:]** → ple Cake’ # Range Slicing

>>> **y[0] + y[1]** → ‘Co’ # Slicing

>>> **len(y)** → 6 # String Length

>>> **s.split(" ")** → [“Apple” , ”Cake”] # split

>>> **Counter(x)** →
Counter({'p': 2, 'e': 2, 'A': 1, 'l': 1, ' ': 1, 'C': 1, 'a': 1, 'k': 1})

>>> **x.lower()** → 'apple cake’

---

x, y = '42', str(42)
>>> **x + y** → ‘4242’ # Concatenate

>>> **x.isnumeric()** → True # is all characters numeric

Boolean

This is a built-in data type that can take up the values: True and False, which often makes them interchangeable with the integers 1 and 0.

Booleans are useful in conditional and comparison expressions

bool() converts a value to a boolean value.

In general, following values are considered False: False, None, 0, empty containers or sequences i.e. empty list, tuple, dictionary, strings etc.

Boolean

Basic Operations

x,y = 2, 2.0

result = 1 < x < 5
>>> **print(result)** → True

>>> **print(x == y)** → True

>>> **print(not(x > y))** → True

>>> **print(x != y)** → False

>>> **not (not x) == x** → True

```
if y % 2 == 0:
   print(y, " is even.")
else:
   print(y, " is odd.")
```

Ans: 2.0 is even.

>>> bool('') → False

>>> **not (x and y) == (not x) | (not y)**
>>> **not (x or y) == (not x) & (not y)**

Data Type Conversion

When you change the type of an entity from one data type to another, this is called "typecasting". There can be two kinds of data conversions possible: implicit termed as coercion and explicit, often referred to as casting.

Implicit Data Type Conversion: This is an automatic data conversion and the compiler handles this for you.

Explicit Data Type Conversion: This type of data type conversion is user defined, which means you have to explicitly inform the compiler to change the data type of certain entities.

Non-Primitive Data Structures

- Arrays
- Lists
- Tuples
- Dictionary
- Sets
- Files

Arrays

Arrays in Python are a compact way of collecting basic data types, all the entries in an array must be of the same data type.

In Python, arrays are supported by the array module and need to be imported before you start initializing and using them. We can try out the array.array when we've numeric data and tight packing along with performance is important.

The elements stored in an array are constrained in their data type. The data type is specified during the array creation and specified using a type code,

Arrays

Basic Operations

import array as arr

a = arr.array('f', (1.0, 1.5, 2.0, 2.5))
>>> **type(a)** → array.array

>>> **a[1:3] -->** array(‘f’, [1.5,2.0])

>>> **a[1] = 23.0** # Arrays are mutable:
Ans: array('f', [1.0, 23.0, 2.0, 2.5])

>>> **del** a[1] → array('f', [1.0, 2.0, 2.5])

>>> **a.append**(42.0) --> array('f', [1.0, 2.0, 2.5, 42.0])

a=arr.array('i',[1,3,2,4,3,5])
>>> a.count(3) → 2

a.pop(2) # drop the element at the position i:
>>> **a** → array('i',[1,3,2,4,3,5])

Lists

Lists in Python are used to store collection of heterogeneous items.

These are mutable, which means that you can change their content without changing their identity.

You can recognize lists by their square brackets [and] that hold elements, separated by a comma,. Lists are built into Python: you do not need to invoke them separately.

Lists

Basic Operations

x = [ ] # Empty list or by x = list()

x1 = [1,2,3,4,5]
>>> **type(x1)** → list

>>> **x1[::-1]** → [5,4,3,2,1]

>>> **x1[::-2]** → ?

>>> **“1” in x1** → False

>>> **x1.sort(reverse = True)** → [5,4,3,2,1]

>>> **list(range(0, 20, 5))** → [0, 5, 10, 15]

>>> **["1"]*3** → [“1” , ”1”, “1”]

x2 = [str(d) for d in x2] # convert all elements to str
>>> **print('-'.join(x2))** → ‘1-orange-3’

lst= [False, False, False, True]
>>> any(lst) → True

Lists

Basic Operations

lst= ["a", "b", "c"]

>>> **lst.append(“e”)** → ["a", "b", "c",”e”]

>>> **lst.insert(3, “d”)** → ['a', 'b', 'c', 'd', 'e']

>>> **lst.remove("d")** → ["a", "b", "c",”e”]

a= [1, 2, 3, 4, 2, 1, 4, 4, 4]

>>> **max(set(a), key = a.count)** → 4 # most frequent

>>> **[x for x in a if x>2]** → [3,4,4,4,4]

Note

Python provides many methods to manipulate and work with lists.

Common list manipulations:

Adding new items to a list: list_num.append (val) | list_num.insert (position, val)
Removing some items from a list: list_num.remove (val) | list_num.pop (position)
Sorting or reversing a list: list.sort (list_num) | list.reverse (list_num)

Arrays vs Lists

Arrays can be very useful when dealing with a large collection of homogeneous data. Arrays are great to do numerical operations on all items individually.	**Lists** are the option to go when you need to deal with non-homogeneous data structures. Arrays need to be declared. Lists don't, since they are built into Python.

Since Python does not have to remember the data type details of each element individually; for some uses arrays may be faster and uses less memory when compared to lists.

Tuples

Tuples are another standard sequence data type, a very similar to list type. The difference between tuples and list is that tuples are immutable, which means once defined you cannot delete, add or edit any values inside it.

Tuples are faster than list. Tuples might also be useful in situations where you might to pass the control to someone else but you do not want them to manipulate data in your collection, but rather maybe just see them or perform operations separately in a copy of the data.

Tuples

Basic Operations

```
x1 = 1,2,3,4,5   # a tuple
x2 = (1,2,3,4,5)  # a tuple
y_tuple = tuple( list(['c', 'a', 'k', 'e', 5]) )
```

>>> **len(x1)** → 5

>>> **x1==x2** --> True

>>> **y_tuple[0]** --> 'c'

>>> **x_tuple[0] = 0** # Cannot change values
Ans:
TypeError: 'tuple' object does not support item assignment

>>> **del x2[1]** # cannot delete values
Ans : **TypeError**: 'tuple' object doesn't support item deletion

>>> **del x2** # delete the whole

Note

If we want to store arbitrary objects, with mixed data types, list or a tuple object can be used.

We can use the built-in str objects to represent textual data as Unicode characters.

Immutable bytes type, or bytearray can come in handy when we want to store contiguous block of bytes.

Sets

Basic Operations

A = {"a", "b", "c"”c”}
B = ["a", "d", "e", “e”]

>>> **A.add(“c”)** → {"a", "b", "c"}

>>> **set(B)** → {'a', 'd', 'e'}

>>> **list(A)** → {“c” , ”a”, “b”}

>>> **A.update(B)** → {'a', 'b', 'c', 'd', 'e'}

>>> **A.remove("d")** → {"a", "b", "c"”e”} # or discard

>>> **“a” in A** → True

>>> **A & B** # intersection

>>> **A | B** # union

>>> **set(('a', 'b', 'b', 'a', 'd'))** → {"a", "b", "d”}

>>> **set("Cookie")** → {'C', 'e', 'i', 'k', 'o'}
>>> **list("Cookie")** → ['C', 'o', 'o', 'k', 'i', 'e']

Dictionary

Basic Operations

car = {"b": "Ford", "y": 1989}

>>> len(car) → 2

>>> car.get("b") → “Ford”

>>> car["y"] = 2019 → {"b": "Ford", "y": 1989}

>>> car[“c”] = “blue”
Ans: {"b": "Ford", "y": 1989, “c”:”blue”}

>>> car.pop("c") → {"b": "Ford", "y": 1989}
>>> car.pop('z', -1) → -1 # instead of error

>>> car.values() # values of dictionary

>>> list(car.keys()) # values of dictionary to list

>>> car.clear() → {} # empty dictionary

```
for x, y in car.items():
   print(x, y)

if "b" in car:
   print("Yes, ‘b’ is one of the keys")
```

Contents

- Data Structures
- **Numpy in Python**
- Algorithms
- Rapid Coding
- Practice Problems

NumPy in Python

NumPy is a python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices.

In Python we have lists that serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is up to 50x faster that traditional Python lists.

NumPy, created in 2005 by Travis Oliphant, is an open source project for free usage.

NumPy in Python

Complex Operations

- Arrays
- Vectors
- Matrices

```
import numpy as np

A= np.array([3, 6., 9, 12])  # 1-dimensional array
B = np.array([(2,4),(6,8)]) # 2-dimensional array
C = np.ones((2,3))  #  2x3 array with all values 1
D = np.full((2,3),5) # 2x3 array with all values 5
E = np.random.rand(2,3)*10 # 2x3  random 0–10
F = np.empty([2, 2], dtype=float) # 2x2  empty

>>> A/3     → [1. 2. 3. 4.]      #Array Division

>>> B/2     → ?

>>> type(A) → <class 'numpy.ndarray'>

>>> A[1:2]) → [6]                #Array Slicing

>>> C.shape → (2,3)

>>> A.dtype → float64

>>> C.T  →   3x2 array  #Transpose

>>> A.reshape(2,2) # Reshape A to 2x2 array
```

NumPy in Python

Complex Operations

- Arrays
- Vectors
- Matrices

```
A = [[1, 0], [0,-2]]
B = np.array([[4, 1], [2, 2]])
>>> np.dot(A, B)                #Matrix Product

Ans: array([[4, 1],
            [-4, -4]])
```

>>> **print(type(A))** → <class 'list'>

>>> **A = np.array(A)** # convert A to numpy array

>>> **np.delete(C,2,axis=0)** # Deletes row #2 of C

>>> **np.delete(C,2,axis=1)** # Deletes column #2

>>> **np.concatenate((A,B),axis=0)** # rows B to A

>>> **np.concatenate((A,B),axis=1)** # cols B to A

>>> **A[1,1]=10** # update element in A

>>> **A[A > 1]** → array([2])

>>> **np.abs(A)** # absolute values

>>> np.floor(A) # Rounds down to the nearest int

Contents

- Data Structures
- Numpy in Python
- **Algorithms**
- Rapid Coding
- Practice Problems

Algorithms

1. **Slicing an array:** Contents of an array object can be accessed and modified by indexing or slicing.

2. **Sorting an array:** Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

Algorithms

Slicing

- A Python slice object is constructed by giving **start**, **stop**, and **step** parameters to the built-in slice function.
- The step allows you to specify slicing (sampling) frequency e.g. slice only every other item.
- This algorithm returns a slice object that can be used used to slice strings, lists, tuple.
- This slice object is usually passed to the array to extract a part of array.

In-built Function

Slicing

slice(start, stop, step)

Get a substring from the given string

py_string = 'Python'

>>> py_string[slice(3)] → 'Pyt'

>>> py_string[:3] → 'Pyt'

>>> py_string[slice(1, 16, 2)] → 'yhn'

Case I

a = ("a", "b", "c", "d", "e", "f", "g", "h")
>>> a[slice(3, 5)] → ???
>>> a[3:5] → ???

Case II

>>>a[slice(0, 8, 3)] → ???

Quick Question

Answer

Case - I

```
>>> a = ("a", "b", "c", "d", "e", "f", "g", "h")
>>> x = slice(3, 5)
>>> print(a[x])
```

Ans: ('d', 'e')

Case - II

```
>>> a = ("a", "b", "c", "d", "e", "f", "g", "h")
>>> x = slice(0, 8, 3)
>>> print(a[x])
```

Ans: ('a', 'd', 'g')

Algorithms

1. **Slicing an array:** Contents of an array object can be accessed and modified by indexing or slicing.

2. **Sorting an array:** Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

Algorithms

Quick Sort

Divide and Conquer algorithm

- The key process in quickSort is partition
- Pick an element as pivot and partitions the given array around the picked pivot
- Given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (< x) before x, and put all greater elements (> x) after x.
- Pivot could be first element, last element, random element or median (implemented here)
- Other sorting algorithms: Merge & Heap Sort
- QuickSort is faster in practice

Algorithms

Sort function in Python

```
# Lists:
 a=[2,3,5,1,7,2]
>>> sorted(a,reverse=True) → [7, 5, 3, 2, 2, 1]
------------------------------
# array:
B = np.array([[4, 1], [2, 2]])
>>> B.sort(axis=0, kind='quicksort')
>>> B[np.argsort(B[:, 1])]


------------------------------
# dictionary
>>> sorted(car.keys())
Dict = {  'e': 72,  'a': 48,  'c': 41}
>>> sorted(Dict .items(), key=lambda x: x[1],
reverse=True)
Ans: [('e', 72), ('a', 48), ('c', 41)]
```

Algorithms

Quick Sort - Pseudocode

```
quickSort(arr[])
{
   pivot = middle element
   left =  array with values lesser than pivot
   middle = array with values  equal to pivot
   right =  array with values greater than pivot

   return ( concatenate(left, middle, right))
}
```

User-defined Function

Quick Sort

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
```

Ans: [1, 1, 2, 3, 6, 8, 10]

Contents

- Data Structures
- Numpy in Python
- Algorithms
- **Rapid Coding**
- Practice Problems

Rapid Coding Tips

1. Make Your Fundamentals Clear. Aim For the Flow.
2. Learn By Doing, Practicing and Not Just Reading.
3. Team Work: Share, Discuss and Ask For Help.
4. Practice Coding as a Team. Soft Skills Matter.
5. Comment and Reflect. Improve Debugging Skills.
6. Create the Right Work Environment. FOCUS.

Some References

1. https://www.w3schools.com/python/python_intro.asp#:~:text=Python%20has%20a%20simple%20syntax,soon%20as%20it%20is%20written.
2. https://www.learnpython.org/
3. https://www.coursera.org/projects/introduction-to-python
4. https://beginnersbook.com/2018/01/introduction-to-python-programming/
5. https://www.slideshare.net/SupunAbeysinghe/preparing-for-ieeextreme-120-amp-mora-xtreme
6. https://github.com/topics/ieeextreme
7. https://www.bigocheatsheet.com/?fbclid=IwAR2NyCWf4L_6hrC8-WhH8vScaeJy_voSZbHSQoH_ByZEsBnijnV8_ueEPEA
8. https://hackerbits.com/data/top-10-data-mining-algorithms-in-plain-english/
9. https://www.datacamp.com/community/tutorials/data-structures-python
10. https://machinelearningmastery.com/linear-algebra-cheat-sheet-for-machine-learning/