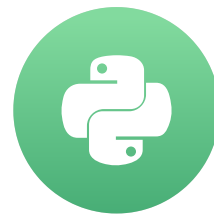


What are the main data structures in Python?

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran



Data Structure

Data Structure - a specialized format to organize and store data.

Main Data Structures in Python:

- list
- tuple
- set
- dictionary

List

list - an ordered mutable sequence of items (e.g. numbers, strings *etc.*)

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

```
[1, 2, 3, 4, 5]
```

List: accessing items

```
my_list = [1, 2, 3, 4, 5]
```

```
print(my_list[2])
```

```
3
```

```
print(my_list[-1])
```

```
5
```

```
print(my_list[1:4])
```

```
[2, 3, 4]
```

```
print(my_list[2:])
```

```
[3, 4, 5]
```

List: modifying items

```
my_list = [1, 2, 3, 4, 5]
```

```
my_list[2] = 30  
print(my_list)
```

```
[1, 2, 30, 4, 5]
```

```
my_list[:2] = [10, 20]  
print(my_list)
```

```
[10, 20, 30, 4, 5]
```

List: methods

```
my_list = [10, 20, 30, 40, 50]
```

```
my_list.append(60)  
print(my_list)
```

```
[10, 20, 30, 40, 50, 60]
```

```
my_list.remove(60)  
print(my_list)
```

```
[10, 20, 30, 40, 50]
```

List: methods

```
my_list = [10, 20, 30, 40, 50]
```

```
my_list.pop()
```

```
50
```

```
print(my_list)
```

```
[10, 20, 30, 40]
```

```
my_list.count(40)
```

```
1
```

Tuple

tuple - an ordered **immutable** sequence of items (e.g. numbers, strings *etc.*)

```
my_tuple = (1, 'apple', 2, 'banana')  
print(my_tuple)
```

```
(1, 'apple', 2, 'banana')
```

```
my_tuple = 1, 'apple', 2, 'banana'  
print(my_tuple)
```

```
(1, 'apple', 2, 'banana')
```


Tuple: modifying values

Modifying items in a tuple is not possible.

```
my_tuple[0] = 10
```

```
TypeError
```

Set

set - an **unordered** collection with no duplicate items (e.g. numbers, strings *etc.*)

```
my_set = set([1, 2, 3, 4, 5])  
print(my_set)
```

```
{1, 2, 3, 4, 5}
```

```
my_set = set([1, 1, 1, 2, 3, 4, 5, 5, 5])  
print(my_set)
```

```
{1, 2, 3, 4, 5}
```

Set: methods

```
my_set1 = set([1, 2, 3, 4, 5])  
my_set2 = set([3, 4, 5, 6, 7])
```

```
my_set1.add(6)  
print(my_set1)
```

```
{1, 2, 3, 4, 5, 6}
```

```
my_set1.remove(6)  
print(my_set1)
```

```
{1, 2, 3, 4, 5}
```

```
my_set1.union(my_set2)
```

```
{1, 2, 3, 4, 5, 6, 7}
```

```
my_set1.intersection(my_set2)
```

```
{3, 4, 5}
```

```
my_set1.difference(my_set2)
```

```
{1, 2}
```

Dictionary

dictionary - a collection of key-value pairs where keys are unique and immutable

key \rightarrow value

```
fruits = {'apple': 10, 'orange': 6, 'banana': 9}
print(fruits)
```

```
{'apple': 10, 'banana': 9, 'orange': 6}
```

```
fruits = dict([('apple', 1), ('orange', 6), ('banana', 9)])
print(fruits)
```

```
{'apple': 10, 'banana': 9, 'orange': 6}
```

Dictionary: accessing values

Accessing a value for a key:

```
fruits = {'apple': 10, 'orange': 6, 'banana': 9}  
fruits['apple']
```

```
10
```

```
fruits['grapefruit']
```

```
KeyError: 'grapefruit'
```

Dictionary: modifying values

```
fruits['apple'] = 20  
print(fruits)
```

```
{'apple': 20, 'orange': 6, 'banana': 9}
```

```
fruits['grapefruit'] = 11  
print(fruits)
```

```
{'apple': 20, 'orange': 6, 'banana': 9, 'grapefruit': 11}
```

Dictionary: methods

```
fruits = {'apple': 10, 'orange': 6, 'banana': 9}
```

```
fruits.items()
```

```
dict_items([('apple', 10), ('orange', 6), ('banana', 9)])
```

Dictionary: methods

```
fruits = {'apple': 10, 'orange': 6, 'banana': 9}
```

```
list(fruits.items())
```

```
[('apple', 10), ('orange', 6), ('banana', 9)]
```


Dictionary: methods

```
fruits = {'apple': 10, 'orange': 6, 'banana': 9}
```

```
fruits.keys()
```

```
dict_keys(['apple', 'orange', 'banana'])
```

```
fruits.values()
```

```
dict_values([10, 6, 9])
```

Dictionary: methods

```
fruits = {'apple': 10, 'orange': 6, 'banana': 9}
```

```
list(fruits.keys())
```

```
['apple', 'orange', 'banana']
```

```
list(fruits.values())
```

```
[10, 6, 9]
```

Dictionary: methods

```
fruits = {'apple': 10, 'orange': 6, 'banana': 9}
```

```
fruits.popitem('banana')
```

```
9
```

```
print(fruits)
```

```
{'apple': 10, 'orange': 6}
```

Operations on Lists, Tuples, Sets, and Dictionaries

```
my_list = [1, 2, 3, 4, 5]  
len(my_list)
```

5

```
my_set = set([1, 2, 3, 4])  
len(my_set)
```

4

```
my_tuple = (1, 2, 3, 4, 5)  
len(my_tuple)
```

5

```
my_dict = {'a': 1, 'b': 2, 'c': 3}  
len(my_dict)
```

3

Operations on Lists, Tuples, Sets, and Dictionaries

```
my_list = [1, 2, 3, 4, 5]  
2 in my_list
```

True

```
my_set = set([1, 2, 3, 4])  
5 in my_set
```

False

```
my_tuple = (1, 2, 3, 4, 5)  
2 in my_tuple
```

True

```
my_dict = {'a': 1, 'b': 2, 'c': 3}  
'b' in my_dict
```

True

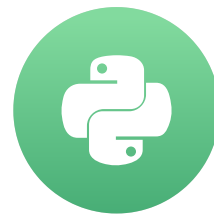
Let's practice!

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

What are common ways to manipulate strings?

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran



String

Create strings:

```
s = 'hello'  
print(s)
```

```
hello
```

```
s = "hello"  
print(s)
```

```
hello
```


String

`str()` constructor:

```
str("hello")
```

```
'hello'
```

```
str(11.5)
```

```
'11.5'
```

```
str([1, 2, 3])
```

```
'[1, 2, 3]'
```

str() constructor

```
class NewClass:  
    def __init__(self, num):  
        self.num = num
```

```
nc = NewClass(2)  
print(nc.num)
```

```
2
```

```
str(nc)
```

```
'<__main__.NewClass instance at 0x105cdabd8>'
```

str() constructor

```
class NewClass:  
    def __init__(self, num):  
        self.num = num  
  
    def __str__(self):  
        return str(self.num)
```

```
nc = NewClass(3)  
str(nc)
```

3

Accessing characters in a string

```
s = "interview"
```

```
s[1]
```

```
'n'
```

```
s[-2]
```

```
'e'
```

```
s[1:4]
```

```
'nte'
```

```
s[2:]
```

```
'terview'
```

```
s[:3]
```

```
'int'
```

The .index() method

```
s = "interview"
```

```
s.index('n')
```

```
1
```

```
s.index('i')
```

```
0
```

Strings are immutable

```
s[0] = 'a'
```

TypeError

```
.capitalize()
```

```
.lower()
```

```
.upper()
```

```
.replace()
```

Methods return a new string object

Modifying methods 1

```
# String concatenation  
s1 = "worm"  
s2 = s1 + "hole"  
print(s2)
```

wormhole

```
# Replace a substring  
s1 = 'a dog ate my food'  
s2 = s1.replace('dog', 'cat')  
print(s2)
```

a cat ate my food

Modifying methods 2

```
# Upper case  
s3 = s2.upper()  
print(s3)
```

```
A CAT ATE MY FOOD
```

```
# Lower case  
s4 = s3.lower()  
print(s4)
```

```
a cat ate my food
```

```
# Capitalization  
s5 = s4.capitalize()  
print(s5)
```

```
A cat ate my food
```


Relation to lists

Create a string from a list of strings:

```
l = ['I', 'like', 'to', 'study']  
s = ' '.join(l)  
print(s)
```

```
I like to study
```

Breaking a string into a list of strings:

```
l = s.split(' ')  
print(l)
```

```
['I', 'like', 'to', 'study']
```

String methods with DataFrames

```
import pandas as pd

d = {'name': ['john', 'amanda', 'rick'], 'age': [35, 29, 19]}
D = pd.DataFrame(d)
print(D)
```

	name	age
0	john	35
1	amanda	29
2	rick	19

String methods with DataFrames

```
D['name'] = # we will modify this column
```

String methods with DataFrames

```
D[ 'name' ] = D[ 'name' ]
```

String methods with DataFrames

```
D['name'] = D['name'].str
```

String methods with DataFrames

```
D[ 'name' ] = D[ 'name' ].str.capitalize()
```

```
print(D)
```

```
   name  age
0  John  35
1 Amanda  29
2  Rick  19
```

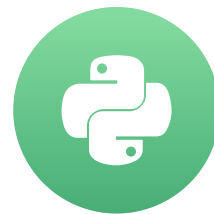
Let's practice!

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

How to write regular expressions in Python?

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON

Kirill Smirnov
Data Science Consultant, Altran



Definition

Regular expression - a sequence of special characters (metacharacters) defining a pattern to search in a text.

cat

"I have a cat. My cat likes to eat a lot. It also catches mice."

Definition

Regular expression - a sequence of special characters (metacharacters) defining a pattern to search in a text.

cat

"I have a **cat**. My **cat** likes to eat a lot. It also **catches** mice."

Complex patterns

Example:

john.smith@mailbox.com is the e-mail of John. He often writes to his boss at boss@big-company.com. But the messages get forwarded to his secretary at info@big-company.com.

Complex patterns

Example:

`john.smith@mailbox.com` is the e-mail of John. He often writes to his boss at `boss@big-company.com`. But the messages get forwarded to his secretary at `info@big-company.com`.

Special characters

Simple characters and numbers are mapped onto themselves:

- `a` → `a`
- `A` → `A`
- `1` → `1`

Dot maps to anything:

- `.` → any character

`.` → `'a'` , `'1'` , `'"'` , `' '` , ...

- `\.` → `.`

Special characters

The following metacharacters represent `\` followed by a letter:

- `\w` → any alphanumeric character or underscore

`\w` → `'1'`, `'a'`, `'_'`, ...

- `\d` → any digit

`\d` → `'1'`, `'2'`, `'3'`, ...

- `\s` → any whitespace character

`\s` → `' '`, `'\t'`, ...

Square brackets

Several metacharacters can be enclosed in square brackets:

- `[aAbB]` \rightarrow `a` , `A` , `b` , `B`
- `[a-z]` \rightarrow `a` , `b` , `c` , ...
- `[A-Z]` \rightarrow `A` , `B` , `C` , ...
- `[0-9]` \rightarrow `0` , `1` , `2` , ...
- `[A-Za-z]` \rightarrow `A` , `B` , `C` , ..., `a` , `b` , `c` , ...

Repetitions

- `*` → no character or it repeats an undefined number of times

`a*` → `''`, `'a'`, `'aa'`, ...

- `+` → the character is present at least once

`a+` → `'a'`, `'aa'`, `'aaa'`, ...

- `?` → the character exists or not

`a?` → `''`, `'a'`

- `{n, m}` → the character is present from `n` to `m` times

`a{2, 4}` → `'aa'`, `'aaa'`, `'aaaa'`

Regular expression for an e-mail

Example:

`john.smith@mailbox.com` is the e-mail of John. He often writes to his boss at `boss@company.com`. But the messages get forwarded to his secretary at `info@company.com`.

```
[ \w\ . ]+@[a-z]+\ . [a-z ]+
```

Regular expression for an e-mail

Example:

`john.smith@mailbox.com` is the e-mail of John. He often writes to his boss at `boss@company.com`. But the messages get forwarded to his secretary at `info@company.com`.

`[\w\ .]+` `@[a-z]+\.[a-z]+`

`[\w\ .]+` → `john.smith` , `boss` , `info`

at least one letter, digit, underscore, or dot character

Regular expression for an e-mail

Example:

`john.smith@mailbox.com` is the e-mail of John. He often writes to his boss at `boss@company.com`. But the messages get forwarded to his secretary at `info@company.com`.

`[\w\ .]+`

`@`

`[a-z]+\.[a-z]+`

`@ → @`

Regular expression for an e-mail

Example:

`john.smith@mailbox.com` is the e-mail of John. He often writes to his boss at `boss@company.com`. But the messages get forwarded to his secretary at `info@company.com`.

`[\w\ .]+@`

`[a-z]+`

`\.[a-z]+`

`[a-z]+`

→

`mailbox`

,

`company`

at least one lowercased letter

Regular expression for an e-mail

Example:

`john.smith@mailbox.com` is the e-mail of John. He often writes to his boss at `boss@company.com`. But the messages get forwarded to his secretary at `info@company.com`.

`[\w\ .]+@[a-z]+` `\.` `[a-z]+`

`\.` \rightarrow `.`

Regular expression for an e-mail

Example:

`john.smith@mailbox.com` is the e-mail of John. He often writes to his boss at `boss@company.com`. But the messages get forwarded to his secretary at `info@company.com`.

`[\w\ .]+@[a-z]+\.`

`[a-z]+`

`[a-z]+` \rightarrow `com`

at least one lowercased letter

re package

```
import re
```

```
pattern = re.compile(r'[\w\.] +@[a-z]+\.[a-z]+')
```

```
text = 'john.smith@mailbox.com is the e-mail of '\n\n'John. He often writes to his boss at '\n\n'boss@company.com. But the messages get forwarded '\n\n'to his secretary at info@company.com.'
```

re.finditer()

```
result = re.finditer(pattern, text)
print(result)
```

```
<callable_iterator object at 0x7f5dff81af98>
```

```
for match in result:
    print(match)
```

```
<_sre.SRE_Match object; span=(0, 22), match='john.smith@mailbox.com'>
<_sre.SRE_Match object; span=(77, 93), match='boss@company.com'>
<_sre.SRE_Match object; span=(146, 162), match='info@company.com'>
```


re.finditer()

```
result = re.finditer(pattern, text)
print(result)
```

```
<callable_iterator object ...>
```

```
for match in result:
    print(match.group())
    print(match.start())
    print(match.end())
```

```
john.smith@mailbox.com
0
22
boss@company.com
77
93
info@company.com
146
162
```

re.findall()

```
substrings = re.findall(pattern, text)
```

```
print(substrings)
```

```
['john.smith@mailbox.com', 'boss@company.com', 'info@company.com']
```

re.split()

```
split_list = re.split(pattern, text)
```

```
print(split_list)
```

```
['',  
 ' is the e-mail of John. He often writes to his boss at ',  
 '. But the messages get forwarded to his secretary at ',  
 '.']
```

Let's practice!

PREPARING FOR CODING INTERVIEW QUESTIONS IN PYTHON